

区块链技术指南



yeasy@github

VISIT…

LANZAROTE
Caliente.com

目錄

前言 1.1
修订记录 1.2
如何贡献 1.3
区块链思想的诞生与概念 1.4
从数字货币说起 1.4.1
比特币：前所未有的社会学实验 1.4.2
区块链核心概念 1.4.3
小结 1.4.4
区块链技术的价值、挑战与展望 1.5
商业价值 1.5.1
关键技术和挑战 1.5.2
趋势与展望 1.5.3
小结 1.5.4
典型应用场景 1.6
金融服务 1.6.1
征信和权属管理 1.6.2
资源共享 1.6.3
投资管理 1.6.4
物联网与供应链 1.6.5
其它场景 1.6.6
小结 1.6.7
分布式系统核心问题 1.7
一致性问题 1.7.1
共识算法 1.7.2
FLP 不可能性原理 1.7.3
CAP 原理 1.7.4
ACID 原则 1.7.5
Paxos 与 Raft 1.7.6
拜占庭问题与算法 1.7.7
可靠性指标 1.7.8

小结 1.7.9
密码学与安全技术 1.8
Hash 算法与摘要 1.8.1
加解密算法 1.8.2
数字签名 1.8.3
数字证书 1.8.4
PKI 体系 1.8.5
Merkle 树 1.8.6
同态加密 1.8.7
其它问题 1.8.8
小结 1.8.9
比特币项目 -- 思想诞生的摇篮 1.9
简介 1.9.1
原理和设计 1.9.2
挖矿 1.9.3
工具 1.9.4
共识机制 1.9.5
闪电网络 1.9.6
侧链 1.9.7
小结 1.9.8
Ethereum（以太坊）-- 技术落地的实验 1.10
简介 1.10.1
安装 1.10.2
相关工具 1.10.3
协议设计 1.10.4
智能合约示例一 1.10.5
小结 1.10.6
Hyperledger（超级账本）-- 商用分布式账本 1.11
诞生与发展 1.11.1
社区组织 1.11.2
五大顶级项目 1.11.3
开发和提交代码 1.11.4
小结 1.11.5
Fabric 部署与管理 1.12

简介 1.12.1
使用 Fabric 1.0 版本 1.12.2
Fabric v0.6 1.12.3
安装部署 1.12.3.1
使用 chaincode 1.12.3.2
权限管理 1.12.3.3
Python 客户端 1.12.3.4
小结 1.12.4
区块链应用开发 1.13
简介 1.13.1
链上代码工作原理 1.13.2
示例一：信息公证 1.13.3
示例二：交易资产 1.13.4
示例三：数字货币发行与管理 1.13.5
示例四：学历认证 1.13.6
示例五：社区能源共享 1.13.7
小结 1.13.8
Fabric 架构与设计 1.14
简介 1.14.1
架构设计 1.14.2
消息协议 1.14.3
小结 1.14.4
区块链服务平台设计 1.15
简介 1.15.1
IBM Bluemix 云区块链服务 1.15.2
微软 Azure 云区块链服务 1.15.3
使用超级账本 Cello 搭建区块链服务 1.15.4
本章小结 1.15.5
性能与评测 1.16
简介 1.16.1
Hyperledger Fabric v0.6 1.16.2
小结 1.16.3
附录 1.17

术语	1.17.1
常见问题	1.17.2
相关组织	1.17.3
ProtoBuf 与 gRPC	1.17.4
参考资源链接	1.17.5

区块链技术指南

1.0.0

区块链技术是金融科技（Fintech）领域的一项重要技术创新。

作为分布式记账（Distributed Ledger Technology，DLT）平台的核心技术，区块链被认为在金融、征信、物联网、经济贸易结算、资产管理等众多领域都拥有广泛的应用前景。区块链技术自身尚处于快速发展的初级阶段，现有区块链系统在设计和实现中利用了分布式系统、密码学、博弈论、网络协议等诸多学科的知识，为学习原理和实践应用都带来了不小的挑战。

本书希望可以探索区块链概念的来龙去脉，剥茧抽丝，剖析关键技术原理，同时讲解实践应用。在开发相关开源分布式账本平台（如超级账本），以及设计基于区块链的企业方案过程中，笔者积累了一些实践经验，也通过本书一并分享出来，希望能推动区块链技术的早日成熟和更多应用场景的出现。

阅读使用

本书适用于对区块链技术感兴趣，且具备一定信息和金融基础知识的读者；无技术背景的读者也可以从中了解到区块链的应用现状。

- 在线阅读：GitBook 或 GitHub
- pdf 版本 下载
- epub 版本 下载

进阶学习

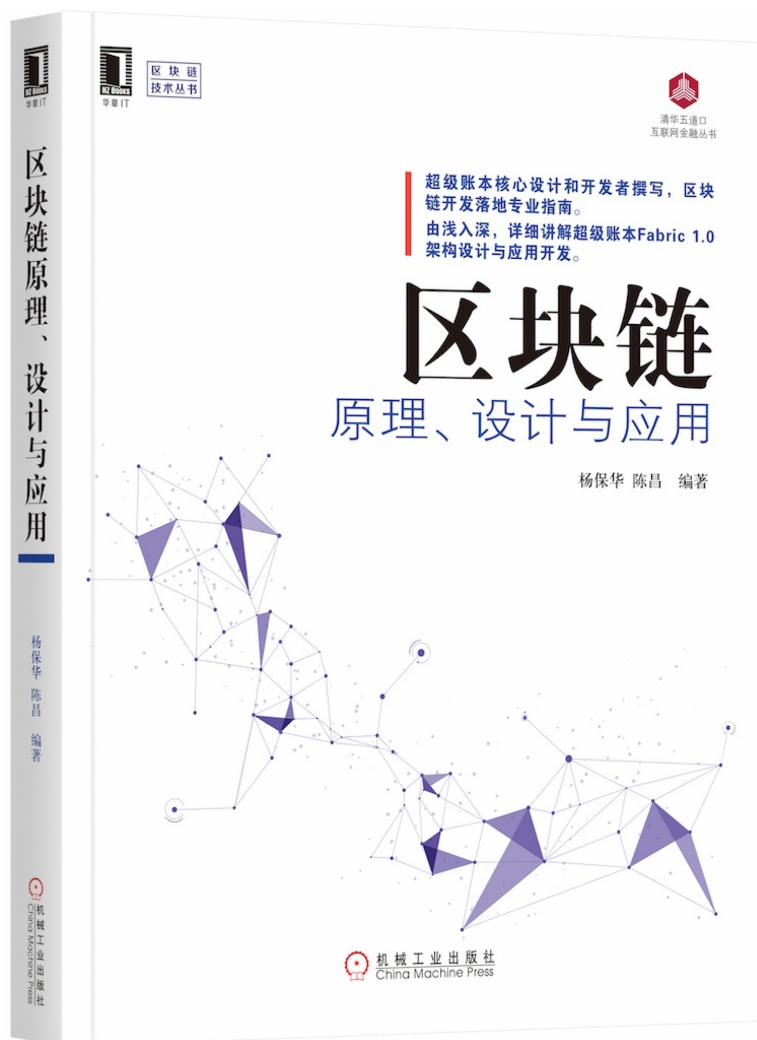


图 *1.1.1* - 区块链原理、设计与应用

《区块链原理、设计与应用》已经正式出版，详细介绍了区块链和分布式账本领域的最新技术，基于超级账本项目介绍了面向企业的分布式账本方案的设计、架构和应用，欢迎大家阅读并反馈建议。

- 京东图书
- 亚马逊图书
- China-Pub
- 当当图书

如果发现疏漏，欢迎提交到 勘误表。

参与贡献

区块链技术自身仍在快速发展中，生态环境也在蓬勃成长。欢迎 参与维护项目。

- 修订记录

- 贡献者名单

鼓励项目

欢迎鼓励项目一杯 coffee~



图 *1.1.2 - coffee*

技术交流

欢迎大家加入区块链技术讨论群：

- QQ 群 I：335626996（已满）
- QQ 群 II：523889325（已满）
- QQ 群 III：414919574（已满）
- QQ 群 IV：364824846（可加）

版本历史

- 1.0.0: 2017-12-31
  - 更新 baas 设计；
  - 更新附录部分；
  - 修正部分表达。
- 0.9.0: 2017-08-24
  - 修正字词；
  - 添加 fabric 1.0 的内容；
  - 《区块链原理、设计与应用》正式出版。
- 0.8.0: 2017-03-07
  - 完善应用场景等；
  - 完善分布式系统技术；
  - 完善密码学技术；
  - 根据最新代码更新 Hyperledger 使用。
- 0.7.0: 2016-09-10
  - 完善一致性技术等；
  - 修正文字。
- 0.6.0: 2016-08-05
  - 修改文字；
  - 增加更多智能合约；
  - 增加更多业务场景。
- 0.5.0: 2016-07-10
  - 增加 Hyperledger 项目的内容；
  - 增加以太坊项目内容；
  - 增加闪电网络介绍、关键技术剖析；
  - 补充区块链即服务；
  - 增加比特币项目。
- 0.4.0: 2016-06-02
  - 添加应用场景分析。
- 0.3.0: 2016-05-12
  - 添加数字货币问题分析。
- 0.2.0: 2016-04-07
  - 添加 Hyperledger 项目简介。

- 0.1.0: 2016-01-17
    - 添加区块链简介。

参与贡献

贡献者 名单。

区块链技术自身仍在快速发展中，生态环境也在蓬勃成长。

本书源码开源托管在 Github 上，欢迎参与维护：github.com/yeasy/blockchain_guide。

首先，在 GitHub 上 `fork` 到自己的仓库，如 `docker_user/blockchain_guide`，然后 `clone` 到本地，并设置用户信息。

```
$ git clone git@github.com:docker_user/blockchain_guide.git
$ cd blockchain_guide
$ git config user.name "yourname"
$ git config user.email "your email"
```

更新内容后提交，并推送到自己的仓库。

```
$ #do some change on the content
$ git commit -am "Fix issue #1: change helo to hello"
$ git push
```

最后，在 GitHub 网站上提交 pull request 即可。

另外，建议定期使用项目仓库内容更新自己仓库内容。

```
$ git remote add upstream https://github.com/yeasy/blockchain_guide
$ git fetch upstream
$ git checkout master
$ git rebase upstream/master
$ git push -f origin master
```

概况

任何事物的发展，从来不是一蹴而就的。

商贸合作中签订的合同，怎么确保对方能遵守和执行？

餐厅宣称刚从海里打捞上来的三文鱼，怎么证明捕捞时间和运输中的卫生？

数字世界里，怎么证明你对资产的所有？

囚徒困境中的两个人，怎样能达成利益的最大化？

宇宙不同文明之间的猜疑链，有没有可能打破？

这些看似很难解决的问题，在区块链的世界里已经有了初步的答案。

本章将简要介绍区块链相关的背景知识，包括其起源、定位、涉及到的关键技术点以及潜在的商业价值。并对区块链的发展进行展望。

从数字货币说起

货币是人类文明发展过程中的一大发明，最重要的职能包括价值尺度、流通手段、贮藏手段。很难想象离开了货币，现代社会庞大而复杂的经济和金融体系还能否持续运转。

历史上，货币的形态经历了多个阶段的演化，包括实物货币、金属货币、代用货币、信用货币、电子货币、数字货币等。货币自身的价值依托也从实物价值、发行方信用价值，到今天出现的对信息系统（包括算法、数学、密码学、软件等）的信任价值。

注：中国最早的关于货币的确切记载“夏后以玄币”出现在恒宽《盐铁论·错币》。

需求

一般等价物都可以作为货币使用。然而平时最常见的还是纸币本位制，既方便携带、不易仿制、又相对容易辩伪。

注意，严格来讲，货币（*money*）不等于现金或通货（*cash*，*currency*），货币的范围更广。

或许有人认为信用卡相对纸币形式更方便。相对于信用卡这样的集中式支付体系来说，货币提供了更好的匿名性。另外，一旦碰到系统故障、断网、没有刷卡机器等情况，信用卡就不可用了。

无论是货币，还是信用卡模式，都需要额外的系统（例如银行）来完成生产、分发、管理等操作，带来很大的额外成本和使用风险。诸如伪造、信用卡诈骗、盗刷、转账等安全事件屡见不鲜。

很自然的，如果能实现一种数字货币，保持既有货币的这些特性，消除纸质货币的缺陷，无疑将带来巨大的社会变革，极大提高经济活动的运作效率。

比较

让我们来对比现在的数字货币和现实生活中的纸币：

属性	分析	胜出方
便携	这点上应该没有争议，显然数字形式的货币胜出。	数字货币
防伪	这点上应该说两者各有千秋，但数字货币可能略胜一筹。纸币依靠的是各种设计（纸张、油墨、暗纹、夹层等）上的精巧，数字货币依靠的则是密码学上的保障。事实上，纸币的伪造时有发生，但数字货币的伪造明面上还没能实现。	数字货币
辩伪	纸币即使依托验钞机仍会有误判情况，数字货币依靠密码学基本不可能出错。数字货币胜出。	数字货币
匿名	通常情况下，两者都能提供很好的匿名性。但都无法防御有意的追踪。	平局
交易	对纸币来说，谁持有纸币就是合法拥有者，交易通过纸币自身的转移即可完成。对数字货币来说则复杂的多，因为任何数字物品都是可以被复制的，因此需要额外的机制。为此，比特币发明了区块链技术来确保可靠不可篡改的交易。	纸币
资源	100 美元钞票的生产成本是 0.1 美元左右。100 面额人民币的生产成本说法众多，但估计应该在几毛到几块范围内。数字货币消耗的资源则复杂的多，以最坏情况估计，算出来多少就要消耗多少电（往往要更多）。	纸币
发行	纸币的发行需要第三方机构的参与，数字货币则通过分布式算法来完成发行。在人类历史上，通胀和通缩往往是不合理地发行货币造成的；数字货币尚无机会被验证，在这方面的表现还有待观察。	平局

可见，数字货币并非在所有领域都优于已有的货币形式。不带前提的在所有领域都鼓吹数字货币并不是一种严谨的态度，应该针对具体情况具体分析。实际上，仔细观察目前支持数字货币的交易机构就会发现端倪，当前还没有一种数字货币能完整起到货币的职能。

最后，虽然当前的数字货币“实验”已经取得了巨大成功，但可见的局限也很明显：其依赖的分布式账本技术还缺乏大规模场景下考验；性能和安全性还有待提升；资源的消耗还过高等等。这些问题还有待于相关技术的进一步发展。

实现挑战

设计和实现一个数字货币并非易事。

在现实生活中，因为纸币具备可转移性，相对容易地完成价值的交割。但是因为电子内容天然具备零复制成本，无法通过发送电子内容来完成价值的转移。持有人可以试图将同一份电子货币发给多个人，这种被称为“双重支付攻击（Double-Spent）”。

也许有人会讲，当前银行中的货币都是电子化的，因为通过账号里面的数字记录了资产。说的没错，这种电子货币模式有人称为“数字货币 1.0”，它实际上是假定存在一个安全可靠的第三方记账机构来实现，这个机构利用信用作为抵押，来完成交易。

这种中心化控制下的数字货币实现相对简单，但需要一个中心管控系统。但是，很多时候并不存在一个安全可靠的第三方记账机构来充当这个中心管控的角色。

例如，贸易两国可能缺乏足够的外汇储备；网络上的匿名双方进行直接买卖；交易的两个机构彼此互不信任，找不到双方都认可的第三方担保；汇率的变化；可能无法连接到第三方的系统；第三方的系统可能会出现故障......

总结一下，在去中心化的场景下，存在几个难题：

- 货币的防伪：谁来负责验证货币；
- 货币交易：如何确定货币从一方转移到另外一方；
- 避免双重支付：如何避免出现双重支付。

好吧，这事其实不太容易。

比特币出现

在不存在一个第三方记账机构的情况下，如何实现一个数字货币系统呢？

近三十年来，数字货币技术朝着这个方向努力，经历了几代演进，包括 e-Cash、HashCash、B-money 等。

1983 年，David Chaum 最早提出ecash，并于 1989 年创建了 Digicash 公司。ecash 系统是首个匿名化的数字加密货币（anonymous cryptographic electronic money, or electronic cash system），基于 David Chaum 发明的盲签名技术，曾被应用于银行小额支付中。ecash 依赖于一个中心化的中介机构，导致它最终失败。

1997 年，Adam Back 发明了 Hashcash，来解决邮件系统中 DoS 攻击问题。Hashcash 首次提出用工作量证明（Proof of Work，PoW）机制来获取额度，该机制后来被后续数字货币技术所采用。

1998 年，Wei Dai 提出了 B-money，将 PoW 引入数字货币生成过程中。B-money 同时是首个面向去中心化设计的数字货币。从概念上看已经比较完善，但是很遗憾的是，其未能提出具体的设计实现。

上面这些数字货币都或多或少的依赖于一个第三方系统的信用担保。直到比特币的出现，将 PoW 与共识机制结合在一起，首次从实践意义上实现了一套去中心化的数字货币系统。

比特币网络无需任何管理机构，自身通过数学和密码学原理来确保了所有交易的成功进行，比特币自身的价值是通过背后的计算力为背书。这也促使人们开始思考在未来的数字世界中，该如何衡量价值，如何发行货币。

目前看来，数字货币比较有影响力的模式有两种，一种是类似 paypal 这样的选择跟已有的系统合作，成为代理；一种是以比特币这样的完全丢弃已有体系的分布式技术。

现在还很难讲哪种模式将成为未来的主流，甚至未来还可能出现更先进的技术。但对比特币这一类数字货币的设计进行探索，将是一件十分有趣的事情。

什么是比特币

历史

2008 年 10 月 31 日，化名 Satoshi Nakamoto （中本聪）的人提出了比特币的设计白皮书（最早见于 metzdowd 邮件列表），并在 2009 年公开了最初的实现代码，第一个比特币是 2009 年 1 月 3 日 18:15:05 生成。但真正流行起来还是在 2010 年后的事情。其官方网站是 bitcoin。

发明人（传言代号为中本聪的澳大利亚人）到目前为止尚无法确认身份，据推测，背后也可能是一个团队。

尽管充满了争议，但从技术角度看，比特币仍然是数字货币历史上一次了不起的创新。比特币网络在 2009 年上线以来已经在全球范围内 `7*24` 小时运行接近 8 年时间，支持过单笔 1.5 亿美金的交易。比特币网络由数千个核心节点参与构成，没有任何中心的运维参与，支持了稳定上升的交易量。

比特币之所以受到无数金融从业者的热捧，在于它首次真正意义上实现了足够安全可靠的去中心化数字货币机制。

作为一种概念金融货币，比特币主要是希望解决已有金融货币系统的几个问题：

- 被掌控在发行机构手中；
- 自身的价值无法保证；
- 无法匿名化交易。

搞金融的人都能想到，实际上，要设计这么一套系统，最关键的还是一套强大的交易记录系统和中立的货币发行机制。

首先，这个系统要能中立、公正、无法被篡改地记录发生过的每一笔交易。对比已有的银行系统，可以看出，现在的银行机制作为第三方，是有代价的提供了这样的服务，即如果交易双方都相信银行的数据库，那么就没问题了。可是如果是世界范围内流通的货币呢？有哪个银行能让大家完全信任它？于是，需要有一套分布式的数据库，在世界范围内都可以访问，而且都无法去控制。这也就是区块链设计的目的。

货币的发行则是通过比特币的协议来规定的，总量必须控制，发行速度会自动调整。既然总量一定，那么单个比特币的价值肯定会随着承认比特币的实体经济的加入而水涨船高。发行速度的调整则避免了通胀或者滞涨的出现。

比特币到区块链

2014 年开始，比特币背后的区块链（Blockchain）技术受到大家关注，并正式引发了分布式记账本（Distributed Ledger）技术的革新浪潮。

人们开始意识到，记账本相关的技术，对于资产（包括有形资产和无形资产）的管理（包括所有权和流通）十分关键；而去中心化的分布式记账本技术，对于当前开放多维化的商业网络意义重大。区块链，正是实现去中心化记账本系统的一种极具潜力的可行技术。

目前，区块链技术已经脱离开比特币，在包括金融、贸易、征信、物联网、共享经济等诸多领域崭露头角。现在当人们提到“区块链”时，往往已经与比特币网络没有直接联系了，除非特别指出是承载比特币交易系统的“比特币区块链”。

什么是区块链

定义

区块链（Blockchain）技术自身仍然在飞速发展中，目前还缺乏统一的规范和标准。

wikipedia 给出的定义为：

> A blockchain —originally, block chain —is a distributed database that maintains a continuously-growing list of data records hardened against tampering and revision. It consists of data structure blocks—which hold exclusively data in initial blockchain implementations, and both data and programs in some of the more recent implementations—with each block holding batches of individual transactions and the results of any blockchain executables. Each block contains a timestamp and information linking it to a previous block.

最早区块链技术雏形出现在比特币项目中。作为比特币背后的分布式记账平台，在无集中式管理的情况下，比特币网络稳定运行了近八年时间，支持了海量的交易记录，并未出现严重的漏洞。

注：比特币历史上唯一已知的漏洞事件曾导致比特币的恶意增发，但问题很快被发现并修正，相关非法交易被撤销。

公认的最早关于区块链的描述性文献是中本聪所撰写的 比特币：一种点对点的电子现金系统，但该文献重点在于讨论比特币系统，实际上并没有明确提出区块链的定义和概念。在其中，区块链被描述为用于记录比特币交易的账目历史。

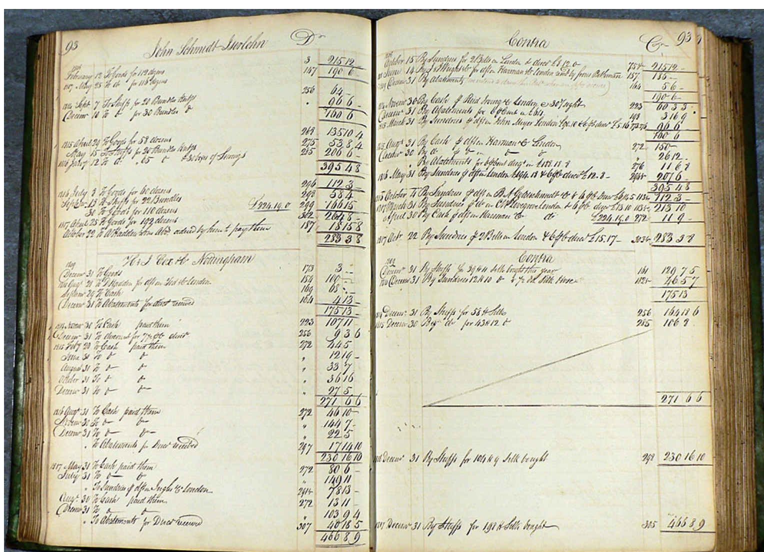


图 *1.4.3.1* - 古老的账本

记账技术历史悠久，现代复式记账系统（Double Entry Bookkeeping）是由意大利数学家卢卡·帕西奥利，1494 年在《Summa de arithmetica, geometrica, proportioni et proportionalità》一书中最早制定。复式记账法对每一笔账目同时记录来源和去向，首次将对账验证功能引入记账过程，提升了记账的可靠性。从这个角度来看，区块链是首个自带对账功能的数字记账技术实现。

更广泛意义地看，区块链属于一种去中心化的记录技术。参与到系统上的节点，可能不属于同一组织、彼此无需信任；区块链数据由所有节点共同维护，每个参与维护节点都能复制获得一份完整记录的拷贝。

跟传统的记账技术相比，其特点应该包括：

- 维护一条不断增长的链，只可能添加记录，而发生过的记录都不可篡改；
- 去中心化，或者说多中心化，无需集中的控制而能达成共识，实现上尽量分布式；
- 通过密码学的机制来确保交易无法抵赖和破坏，并尽量保护用户信息和记录的隐私性。

更进一步的，还可以将智能合约跟区块链结合到一起，让其提供除了交易（比特币区块链已经支持简单的脚本计算）功能外更灵活的合约功能，执行更为复杂的操作。这样扩展之后的区块链，已经超越了单纯数据记录的功能了，实际上带有点“普适计算”的意味了。

从技术特点上，可以看到现在区块链技术的三种典型应用场景：

定位	功能	智能合约	一致性	权限	类型	性能	代表
公信的数字货币	记账功能	不带有或较弱	PoW	无	公有链	较低	比特币
公信的交易处理	智能合约	图灵完备	PoW、PoS	无	公有链	受限	以太坊
带权限的交易处理	商业处理	多种语言，图灵完备	多种，可插拔	支持	联盟链	可扩展	Hyperledger

基本原理

区块链的基本原理理解起来并不难。基本概念包括：

- 交易（Transaction）：一次操作，导致账本状态的一次改变，如添加一条记录；
- 区块（Block）：记录一段时间内发生的交易和状态结果，是对当前账本状态的一次共识；
- 链（Chain）：由一个个区块按照发生顺序串联而成，是整个状态变化的日志记录。

如果把区块链作为一个状态机，则每次交易就是试图改变一次状态，而每次共识生成的区块，就是参与者对于区块中所有交易内容导致状态改变的结果进行确认。

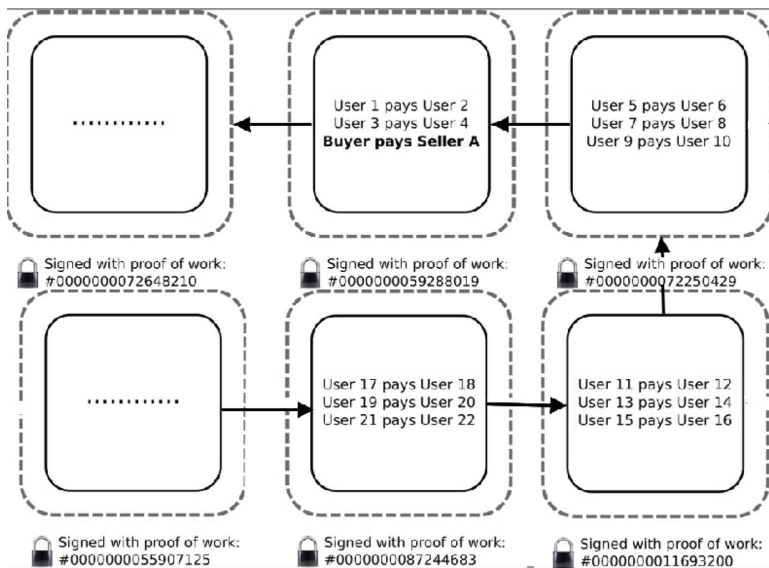


图 *1.4.3.2* - 区块链示例

在实现上，首先假设存在一个分布式的数据记录本（这方面的技术相对成熟），这个记录本只允许添加、不允许删除。其结构是一个线性的链表，由一个个“区块”串联组成，这也是其名字“区块链”的来源。新的数据要加入，必须放到一个新的区块中。而这个块（以及块里的交易）是否合法，可以通过一些手段快速检验出来。维护节点都可以提议一个新的区块，然而必须经过一定的共识机制来对最终选择的区块达成一致。

具体以比特币为例来看如何使用了区块链技术？客户端发起一项交易后，会广播到网络中并等待确认。网络中的节点会将一些等待确认的交易记录打包在一起（此外还要包括此前区块的哈希值等信息），组成一个候选区块。然后，试图找到一个 nonce 串放到区块里，使得候选区块的 hash 结果满足一定条件（比如小于某个值）。一旦算出来这个区块在格式上就合法了，就可以进行全网广播。大家拿到提案区块，进行验证，发现确实符合约定条件了，就承认这个区块是一个合法的新区块，被添加到链上。当然，在实现上还会有很多的细节。

比特币的这种基于算力的共识机制被称为 Proof of Work（PoW）。目前，要让 hash 结果满足一定条件并无已知的启发式算法，只能进行暴力尝试。尝试的次数越多，算出来的概率越大。通过调节对 hash 结果的限制，比特币网络控制约 10 分钟平均算出来一个合法区块。算出来的节点将得到区块中所有交易的管理费和协议固定发放的奖励费（目前是 12.5 比特币，每四年减半）。也即俗称的挖矿。

很自然会有人问，能否进行恶意操作来破坏整个区块链系统或者获取非法利益。比如不承认别人的结果，拒绝别人的交易等。实际上，因为系统中存在大量的用户，而且用户默认都只承认他看到的最长的链。只要不超过一半（概率意义上越少肯定越难）的用户协商，最终最长的链将很大概率上是合法的链，而且随着时间增加，这个概率会越大。例如，经过 6 个块后，即便有一半的节点联合起来想颠覆被确认的结果，其概率将为 ，即低于 的可能性。

注：熟悉 *Git* 的人，应该会赞叹两者在设计上的异曲同工之妙。

分类

根据参与者的不同，可以分为公开（Public）链、联盟（Consortium）链和私有（Private）链。

公开链，顾名思义，任何人都可以参与使用和维护，典型的如比特币区块链，信息是完全公开的。

如果引入许可机制，包括私有链和联盟链两种。

私有链，则是集中管理者进行限制，只能得到内部少数人可以使用，信息不公开。

联盟链则介于两者之间，由若干组织一起合作维护一条区块链，该区块链的使用必须是有权限的管理，相关信息会得到保护，典型如银联组织。

目前来看，公开链将会更多的吸引社区和媒体的眼球，但更多的商业价值应该在联盟链和私有链上。

根据使用目的和场景的不同，又可以分为以数字货币为目的的货币链，以记录产权为目的的产权链，以众筹为目的的众筹链等。

误区

目前，对区块链的认识还存在不少误区。

首先，区块链不是数据库。虽然区块链也可以用来存储数据，但它要解决的问题是多方的互信问题。单纯从存储数据角度，它的效率可能不高，笔者也不推荐把大量的原始数据放到区块链上。

其次，区块链不是要颠覆现有技术。作为基于多项已有技术而出现的新事物，区块链跟现有技术的关系是一脉相承的，在解决多方合作和可信处理上多走了一步，但并不意味着它将彻底颠覆已有的商业模式。很长一段时间里，区块链的适用场景仍需摸索，跟已有系统必然是合作共存的关系。

小结

区块链是第一个试图自带信任化和防止篡改的分布式记录系统。它的出现，让大家意识到，除了互联网这样的尽力而为的基础设施外，我们还能打造一个彼此信任的基础设施。

类似比特币这样的大规模长时间自治运行的系统，也为区块链技术的应用开启了更多遐想的空间。如果人与人之间的交易无法伪造，合同都能确保可靠执行，世界是不是更美好一些了呢？这是技术进步再次给人类发展带来福利。

不提这种去中心化的金融系统是否能在现实中普及，在跨国交易、跨组织合作日益频繁的今天，已经有了不少有意义的尝试和参考。

更进一步，比特币只是基于区块链技术的一种金融应用（而且是直接嵌入区块链中），区块链技术还能带来更通用的计算能力。Hyperledger 和 Ethereum 就试图做类似的事情，基于区块链再做一层平台层，让别人基于平台开发应用变得更简单。

另外，区块链本身可以作为分布式存储，也自然可以作为分布式计算引擎。可以想象，整个加入集群的设备都是计算引擎，大家通过付费来使用计算力，是不是就有点普适计算的意味了？

有理由相信，随着更多商业应用场景的出现，区块链技术将在未来金融和信息技术领域占据一席之地。

概况

任何事物的发展，从来不是一蹴而就的。

商贸合作中签订的合同，怎么确保对方能遵守和执行？

餐厅宣称刚从海里打捞上来的三文鱼，怎么证明捕捞时间和运输中的卫生？

数字世界里，怎么证明你对资产的所有？

囚徒困境中的两个人，怎样能达成利益的最大化？

宇宙不同文明之间的猜疑链，有没有可能打破？

这些看似很难解决的问题，在区块链的世界里已经有了初步的答案。

本章将简要介绍区块链相关的背景知识，包括其起源、定位、涉及到的关键技术点以及潜在的商业价值。并对区块链的发展进行展望。

商业价值

现代商业的典型模式为，交易方通过协商和执行合约，完成交易过程。区块链擅长的正是如何管理合约，确保合约的顺利执行。

根据类别和应用场景不同，区块链所体现的特点和价值也不同。

从技术特点上，区块链一般被认为具有：

- 分布式容错性：网络极其鲁棒，容错 1/3 左右节点的异常状态。
- 不可篡改性：一致提交后的数据会一直存在，不可被销毁或修改。
- 隐私保护性：密码学保证了未经授权者能访问到数据，但无法解析。

随之带来的业务特性将可能包括：

- 可信任性：区块链技术可以提供天然可信的分布式账本平台，不需要额外第三方中介机构。
- 降低成本：跟传统技术相比，区块链技术可能带来更短的时间、更少的人力和维护成本。
- 增强安全：区块链技术将有利于安全可靠的审计管理和账目清算，减少犯罪可能性，和各种风险。

区块链并非凭空诞生的新技术，更像是技术演化到一定程度突破应用阈值后的产物，因此，其商业应用场景也跟促生其出现的环境息息相关。基于区块链技术，任何基于数字交易的活动成本和追踪成本都会降低，并且能提高安全性。笔者认为，能否最终带来成本的降低，将是一项技术能否被深入应用的关键。

笔者认为，所有跟信息、价值（包括货币、证券、专利、版权、数字商品、实际物品等）、信用等相关的交换过程，都将可能从区块链技术中得到启发或直接受益。但这个过程绝不是一蹴而就的，可能经过较长时间的探索和论证

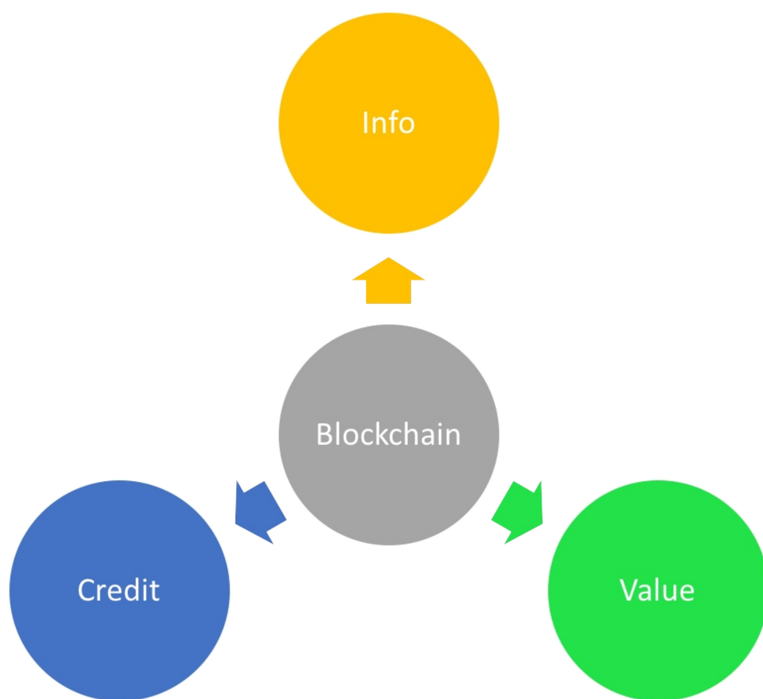


图 *1.5.1.1* - 区块链影响的交换过程

目前，区块链技术已经得到了众多金融机构和商业公司的关注。

已经对区块链技术进行投入或应用的金融机构（排名不分先后）目前有：

- Visa
- 美国纳斯达克证券交易所（Nasdaq）
- 高盛投资银行（Goldman Sachs）
- 花旗银行（citibank）
- 美国富国银行（Wells Fargo）
- 中国央行
- 中国浦发银行
- 日本三菱日联金融集团
- 瑞士联合银行
- 德意志银行
- DTCC
- 全球同业银行金融电讯协会（SWIFT）

部分商业、技术公司包括：

- IBM
- 微软
- Intel
- 思科（Cisco）
- 埃森哲

关键技术和挑战

从技术角度讲，区块链涉及到的领域比较杂，包括分布式、存储、密码学、心理学、经济学、博弈论、网络协议等，下面列出了目前认为有待解决或改进的关键技术点。

密码学技术

怎么防止交易记录被篡改？

怎么证明交易方的身份？

怎么保护交易双方的隐私？

密码学正是要提供解决这些问题的有效手段。传统方案包括 hash 算法，加解密算法，数字证书和签名(盲签名、环签名)等。区块链技术的应用将可能刺激密码学的进一步发展，包括随机数的产生、安全强度、加解密处理的性能等。量子计算等新技术的出现，让 RSA 算法等已经无法提供足够的安全性。

这将依赖于数学科学的进一步发展和新一代计算技术的突破。

注：*SONY PS3 私钥被破解事件* 再次证明，即便足够安全的算法，如果没有被恰当的使用，都只是纸上谈兵。

分布式共识

这是个古老的话题，已有大量的研究成果（Paxos、拜占庭等）。

核心在于如何解决某个变更在网络中是一致的，是被大家都承认的，同时这个信息是被确定的，不可推翻的。该问题在公开匿名场景下和带权限管理的场景下需求差异较大。

比特币区块链考虑的是公开匿名场景下的最坏保证。引入了“工作量证明”（Proof of Work）策略来规避少数人恶意破坏数据，并通过概率模型保证最后大家看到的就是合法的最长链。此外，还有以权益为抵押的 PoS、DPoS 和 Casper 等。这些算法在思想上都是基于经济利益的博弈，让恶意破坏的参与者损失经济利益，从而保证大部分人的合作。同时，确认必须经过多个区块的生成之后从概率学上进行保证。

更广泛的区块链技术支持更多的共识机制，包括经典的拜占庭算法等，可以解决确定性的问题。

共识问题在很长一段时间内都将是极具学术价值的研究热点，核心的指标将包括容错的节点比例和收敛速度。PoW 等系列算法理论上允许少于一半的不合作节点，PBFT 等算法理论上允许不超过 的不合作节点。

处理性能

如何提高交易的吞吐量，同时降低交易的确认延迟。

目前，公开的比特币区块链只能支持平均每秒约 7 笔的吞吐量，一般认为对于大额交易来说，安全的交易确认时间为一个小时。小额交易只要确认被广播到网络中并带有交易服务费用，即有较大概率被最终打包到区块中。

区块链系统跟传统分布式系统不同，其处理性能无法通过单纯增加节点数来进行扩展，实际上，很大程度上取决于单个节点的处理能力。高性能、安全、稳定性、硬件辅助加解密能力，都将是考察节点性能的核心要素。

一方面可以将单个节点采用高性能的处理硬件，同时设计优化的策略和算法，提高性能；另外一方面将大量高频的交易放到链外来，只用区块链记录最终交易信息，如 闪电网络 等。类似的，侧链（side chain）、影子链（shadow chain）等的思路在当前阶段也有一定的借鉴意义。类似设计可以很容易的将交易性能提升 1-2 个数量级。此外，如果采用联盟链的方式，在一定的信任前提和利益约束下优化设计，也可以换来性能的提升。

目前，开源区块链自身在平台层面已经实现普通配置，单客户端每秒数百次的交易吞吐量（参考后面的 性能评测数据），乐观预测将很快突破每秒数千次的基准线，但离现有证券交易系统的每秒数万笔的峰值还是有较大差距。

另外，从工程设计和平台部署上，都存在一些可以优化的地方。

注：*VISA* 系统的处理均值为 *2000 tps*，号称的峰值为 *56,000 tps*；某支付系统的处理峰值超过了 *85,000 tps*；某证券交易所号称的处理均（峰）值在 *80,000 tps* 左右。

扩展性

常见的分布式系统，可以通过增加节点来扩展整个系统的处理能力。

对于区块链网络系统来说，这个问题并非那么简单。

网络中每个参与维护的核心节点都要保持一份完整的存储，并且进行智能合约的处理。因此，整个网络的总存储和计算能力，取决于单个节点。甚至当网络中节点数过多时，可能会因为一致性的达成过程延迟降低整个网络的性能。尤其在公有网络中，由于大量低质量处理节点的存在问题将更明显。

比较直接的一些思路，是放松对每个节点都必须参与完整处理的限制（但至少部分节点要能合作完成完整的处理），这个思路已经在超级账本中启用；同时尽量减少核心层的处理工作。

在联盟链模式下，还可以专门采用高性能的节点作为核心节点，用相对较弱的节点作为代理访问节点。

系统安全

区块链目前最热门的应用前景是金融相关的服务，安全自然是讨论最多、挑战最大的话题。

区块链在设计上基于现有的成熟的密码学算法。但这是否就能确保其安全呢？

世界上并没有绝对安全的系统。

系统是由人设计的，系统也是由人来运营的，只要有人参与的系统，就容易出现漏洞。

可以参考，著名黑客米特尼克所著的《反欺骗的艺术——世界传奇黑客的经历分享》，介绍了大量的实际社交工程欺骗场景。

有如下几个方面是很难逃避的。

首先是立法。对区块链系统如何进行监管？攻击区块链系统是否属于犯罪？攻击银行系统是要承担后果的。但是目前还没有任何法律保护区块链以及基于它的实现。

其次是软件实现的潜在漏洞是无法避免的。考虑到使用了几十年的 openssl 还带着那么低级的漏洞（heart bleeding），而且是源代码就在大家眼皮底下。这背后曾经发生过啥，让人遐想连篇。对于金融系统来说，无论客户端还是平台侧，即便是很小的漏洞都可能造成难以估计的损失。

另外，公有区块链所有交易记录都是公开可见的。搞大数据的人听了是不是开始激动起来了，确实，这里面能分析的东西还真不少，而且规模够大、影响力够大……实际上，已有文献证明，比特币区块链的交易记录最终是能追踪到用户的。

还有就是作为一套完全的分布式系统，公有的区块链缺乏有效的调整机制，一旦运行起来，出现问题也难以修正。即使是让它变得更公平、更完善的修改，只要有部分既得利益者合起来反对，那就无法加入进去。这让比特币本身的价值也蒙上了一层阴影。

此外，运行在区块链上的智能合约应用可能是五花八门的，必须要有办法进行安全管控，在注册和运行前需要有机制进行探测，以规避恶意代码的破坏。

2016 年 6 月 17 日，发生 DAO 系统漏洞被利用 事件，直接导致价值 6000 万美元的数字货币被利用者获取。尽管对于这件事情的反思还在进行中，但事实再次证明，目前基于区块链技术进行生产应用时，务必要细心谨慎地进行设计和验证。

数据库和存储系统

区块链网络中的块信息需要写到数据库中进行存储。

观察区块链的应用，大量的写操作、hash 计算和验证操作，跟传统数据库的行为十分不同。

当年，人们观察到互联网应用大量非事务性的查询操作，而设计了非关系型（NoSql）数据库。那么，针对区块链应用的这些特点，是否可以设计出一些特殊的针对性的数据库呢？

levelDB、RocksDB 等键值数据库，具备很高的随机写和顺序读\/写性能，以及相对较差随机读的性能，被广泛应用到了区块链信息存储中。但目前来看，面向区块链的数据库技术仍然是需要突破的技术难点之一。

笔者认为，未来将可能出现更具针对性的“块数据库（BlockDB）”，专门服务类似区块链这样的新型数据业务，其中每条记录将包括一个完整的区块信息，并天然地跟历史信息进行关联，一旦写入确认无法修改。所有操作的最小单位将是一个块。

可集成性

在相当长的一段时间内，基于区块链的新业务系统将与已有的中心化系统共存。

两种系统如何共存，如何分工，彼此的业务交易如何进行传递？

这些都是很迫切的问题。这个问题解决不好，将是区块链技术落地的很大阻碍。

其它

区块链提供的新应用和新的业务场景，也带来了对很多具体的运营问题。

例如：

- 智能合约的合法性、安全性和可执行性；
- 如何将现实中的合约和条约对应为电子合约；
- 分布式系统的伸缩可靠性和数据迁移；
- 对存储系统新的挑战，特别是性能。

趋势与展望

关于区块链的探讨和争论，自其诞生之日起就从未停息。

或许从计算技术的演变历史中能得到一些启发。

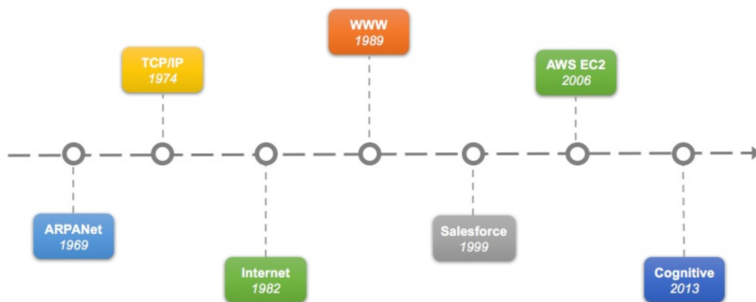


图 *1.5.3.1* - 计算的历史，笔者于某次技术交流会中提出

以云计算为代表的现代计算技术，其发展历史上有若干重要的时间点和事件：

- 1969 - ARPANet（Advanced Research Projects Agency Network）：现代互联网的前身，被美国高级研究计划署（Advanced Research Project Agency）提出，其使用 NCP 协议，核心缺陷之一是无法做到和个别计算机网络交流；
- 1973 - TCP/IP：Vinton.Cerf（文特•瑟夫）与Bob Karn（鲍勃•卡恩）共同开发出 TCP 模型，解决了 NCP 的缺陷；
- 1982 - Internet：TCP/IP 正式成为规范，并被大规模应用，现代互联网诞生；
- 1989 - WWW：早期互联网的应用主要包括 telnet、ftp、email 等，蒂姆·伯纳斯-李（Tim Berners-Lee）设计的 WWW 协议成为互联网的杀手级应用，引爆了现代互联网，从那开始，互联网业务快速扩张；
- 1999 - salesforce：互联网出现后，一度只能进行通信应用，但 salesforce 开始以云的理念提供基于互联网的企业级服务；
- 2006 - aws ec2：AWS EC2 奠定了云计算的业界标杆，直到今天，竞争者们仍然在试图追赶 AWS 的脚步；
- 2013 - cognitive：以 IBM Watson 为代表的认知计算开始进入商业领域，计算开始变得智能，进入“后云计算时代”。

从这个历史中能看出哪些端倪呢？

一个是 技术领域也存在着周期律。 这个周期目前看是 7-8 年左右。或许正如人有“七年之痒”，技术也存在着七年这道坎，到了这道坎，要么自身突破迈过去，要么往往就被新的技术所取代。如果从比特币网络上线（2009 年 1 月）算起，到今年正是在坎上。因此，现在正是相关技术进行突破的好时机。

为何恰好是 *7* 年？*7* 年按照产品周期来看基本是 *2~3* 个产品周期，所谓事不过三，经过 *2~3* 个产品周期也差不多该有个结论了。

另外，最早出现的未必是先驱，也可能是先烈。 创新固然很好，但过早播撒的种子，没有合适的土壤，往往也难长大。技术创新与科研创新很不同的一点便是，技术创新必须立足于需求，过早过晚都会错失良机。科研创新则要越早越好，最好像二十世纪那批物理巨匠们一样，让后人吃了一百多年的老本。

最后，事物的发展往往是延续的、长期的。 新生事物大都不是凭空蹦出来的，往往是解决了前辈未能解决的问题，或是出现了之前未曾出现过的场景。而且很多时候，新生事物会在历史的舞台下面进行长期的演化，只要是往提高生产力的正确方向，迟早会出现在舞台上的一天。

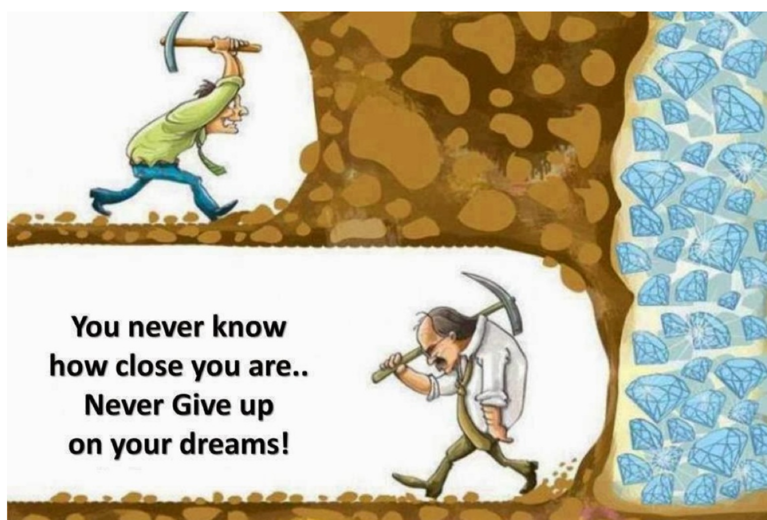


图 *1.5.3.2* - 坚持还是放弃？

目前，区块链在数字货币领域（以比特币为代表）的应用已经相对成熟，而在智能合约方向尚处于初步实践阶段。区块链技术的应用已经在许多领域都带来了生产力提升，笔者相信，随着技术进一步的发展，区块链将会促进金融和信息科技走向新的阶段。

小结

区块链是第一个试图自带信任化和防止篡改的分布式记录系统。它的出现，让大家意识到，除了互联网这样的尽力而为的基础设施外，我们还能打造一个彼此信任的基础设施。

类似比特币这样的大规模长时间自治运行的系统，也为区块链技术的应用开启了更多遐想的空间。如果人与人之间的交易无法伪造，合同都能确保可靠执行，世界是不是更美好一些了呢？这是技术进步再次给人类发展带来福利。

不提这种去中心化的金融系统是否能在现实中普及，在跨国交易、跨组织合作日益频繁的今天，已经有了不少有意的尝试和参考。

更进一步，比特币只是基于区块链技术的一种金融应用（而且是直接嵌入区块链中），区块链技术还能带来更通用的计算能力。Hyperledger 和 Ethereum 就试图做类似的事情，基于区块链再做一层平台层，让别人基于平台开发应用变得更简单。

另外，区块链本身可以作为分布式存储，也自然可以作为分布式计算引擎。可以想象，整个加入集群的设备都是计算引擎，大家通过付费来使用计算力，是不是就有点普适计算的意味了？

有理由相信，随着更多商业应用场景的出现，区块链技术将在未来金融和信息技术领域占据一席之地。

应用场景

应用为王。

一项技术能否最终存活下来，有很多决定因素，但其中十分关键的便是是否能找到合适的应用场景。

区块链最近几年炒得很热，国内已有大量与之相关的企业，有些企业已经结合已有业务摸索出了自己的应用场景，但仍有不少企业处于不断试探和反复迷惑状态。

实际上，要找到合适的应用场景，还是要从区块链自身的特性出发进行分析。区块链在不引入第三方中介机构的前提下，可以提供去中心化、不可篡改、安全可靠等特性保证。因此，所有直接或间接依赖于第三方担保信任机构的活动，均可能从区块链技术中获益。

笔者认为，未来几年内，可能深入应用区块链的场景将包括：

- 金融服务：主要是降低交易成本，减少跨组织交易风险等。该领域的区块链应用将最快成熟起来，银行和金融交易机构将是主力推动者。
- 征信和权属管理：这是大型社交平台和保险公司都梦寐以求的，目前还缺乏足够的数据来源、可靠的平台支持和有效的数据分析和管理。该领域创业的门槛极高，需要自上而下的推动。
- 资源共享：airbnb 为代表的公司将欢迎这类应用，极大降低管理成本。这个领域创业门槛低，主题集中，会受到投资热捧。
- 投资管理：无论公募还是私募基金，都可以应用区块链技术降低管理成本和管控风险。虽然有 DAO 这样的试水，谨慎认为该领域的需求还未成熟。
- 物联网与供应链：物联网是很适合的一个领域，短期内会有大量应用出现，特别是租赁、物流等特定场景。但物联网自身的发展局限将导致短期内较难出现规模应用。

当然，短期内部分场景可能还难以实现，但区块链技术的正确应用会促进这些行业的进一步发展。

金融服务

自有人类社会以来，金融交易就是必不可少的经济活动。交易角色和内容的不同，反映出来就是不同的生产关系。通过交易，可以优化社会的效率，实现价值的最大化。人类社会的发展，离不开交易形式的演变。可见，交易在人类社会中的地位有多重要。

交易本质上交换的是价值的所属权。现在为了完成交易（例如房屋、车辆的所属权），往往需要一些中间环节，特别是中介担保角色。这是因为，交易双方往往存在着不充分信任的情况，要证实价值所属权并不容易，而且往往彼此的价值不能直接进行交换。合理的中介担保，确保了交易的正常运行，提高了经济活动的效率，但已有的第三方中介机制往往存在成本高、时间周期长、流程复杂、容易出错等缺点。正是因为这些，金融服务成为区块链最为火热的应用领域之一。

区块链技术可以为金融服务提供有效可靠的所属权证明和相当强的中介担保机制。

金融服务涉及的领域包括货币、证券、保险、捐赠等。

银行金融管理

银行分为中央银行和普通银行。

中央银行的两大职能是“促进宏观经济稳定”和“维护金融稳定”（《金融的本质》，伯克南），主要手段就是管理各种证券和利率。央行的存在，为整个社会的金融体系提供了最终的信用担保。

普通银行则往往基于央行的信用，实际作为中介担保，来协助完成多方的金融交易。

银行的活动包括发行货币，完成存款、贷款等大量的交易内容。银行必须能够确保交易的确定性，必须通过诸多手段确立自身的信用地位。

传统的金融系统为了完成上述功能，开发了极为复杂的软件和硬件方案，不仅消耗了昂贵的成本，还需要大量的维护成本。即便如此，这些系统仍然存在诸多缺陷，例如很多交易都不能在短时间内完成，每年发生大量的利用银行相关金融漏洞进行的犯罪。

此外，在目前金融系统流程情况下，大量商家为了完成交易，还常常需要额外的组织（如支付宝）进行处理，这些实际上都增加了目前金融交易的成本。

区块链技术被认为是有可能促使这一行业发生革命性变化的“奇点”。除了众所周知的比特币等数字货币之外，还有诸多金融机构进行了有意义的尝试。

欧洲央行评估区块链在证券交易后结算的应用

目前，全球交易后的对账和处理费用超过 200 亿美金。

欧洲央行的 报告 显示，区块链作为分布式账本技术，可以节约对账的成本，同时让证券所有权的变更可能变得近乎实时。

中国中央银行投入区块链研究

央行行长周小川曾表示央行数字货币可能将采用区块链模式，彻底改变传统货币流通模式。据悉，已有专门的团队在进行评估和实践。

2016 年 1 月 20 日，专门组织了“数字货币研讨会”，邀请了花旗、德勤等公司的区块链专家就数字货币发行的总体框架、演进、国家加密货币等话题进行了研讨。

会后，发布对我国银行业数字货币的战略性发展思路，提出要早日发行数字货币，并利用数字货币相关技术来打击金融犯罪活动。

加拿大银行提出新的数字货币

加拿大央行正在开发基于区块链技术的数字版加拿大元（名称为 CAD 币），以允许用户可以使用加元来兑换该数字货币。经过验证的对手方将会处理交易，如果需要，银行将保留销毁CAD币的权利。

发行 CAD 币是更大的一个探索型科技项目 Jasper 的一部分。据悉，除了加拿大央行外，蒙特利尔银行、加拿大帝国商业银行、加拿大皇家银行、加拿大丰业银行、多伦多道明银行等多家机构也都参与了该项目。

来源：金融时报- Canada experiments with digital dollar on blockchain，2016-06-16。

英国银行实现 RSCoin

英国银行在数字化货币方面进展十分突出，已经实现了基于分布式账本平台的数字化货币系统，RSCoin。

RSCoin 目标是提供一个由中央银行控制的数字货币，采用了双层链架构、改进版的 2PC 提交，以及多链条之间的交叉验证机制。因为主要是央行和下属银行之间使用，通过提前建立一定的信任基础，可以提供较好的处理性能。

中国邮储银行将区块链技术应用到核心业务系统

2016年10月，中国邮储银行宣布携手 IBM 推出基于区块链技术的资产托管系统，为中国银行业首次将区块链技术应成功用于核心业务系统。

新的业务系统免去了重复的信用校验过程，将原有业务环节缩短了约60-80%，提高了信用交易的效率。

各种新型支付业务

基于区块链技术，出现了大量的创新支付企业。

- Abra：区块链数字钱包，无需银行账户，实现不同币种的兑换。
- Bitwage：基于比特币区块链的跨境工资支付平台。
- BitPOS：低成本的快捷线上支付。
- Circle：由区块链充当支付网络，允许用户快速进行跨币种的快速汇款。
- Ripple：实现跨境的多币种低成本实时交易，引入了网关概念（类似银行）。

证券交易

证券交易包括交易执行和确认环节。

交易本身相对简单，主要是由交易系统（极为复杂的软硬件系统）完成电子数据库中内容的变更。中心的验证系统极为复杂和昂贵；交易指令执行后的结算和清算环节也十分复杂，往往需要较多人力成本和大量的时间，并且容易出错。

目前来看，基于区块链的处理系统还难以实现海量交易系统所需要的性能（每秒一万笔以上成交，日处理能力超过五千万笔委托、三千万笔成交）。但在交易的审核和清算环节，区块链技术存在诸多的优势，可以避免人工的参与。

> 咨询公司 Oliver Wyman 给 SWIFT(环球同业银行金融电讯协会)提供的研究报告预计全球清算行为成本约 50~100 亿美元,结算成本、托管成本和担保物管理成本 400~450 亿美元(390 亿美元为托管链的市场主体成本)，而交易后流程数据及分析花费 200~250 亿美元。

2015 年 10 月，美国纳斯达克（Nasdaq）证券交易所推出区块链平台 Nasdaq Linq，实现主要面向一级市场的股票交易流程。通过该平台进行股票发行的的发行者将享有“数字化”的所有权。

其它相关企业还包括：

- BitShare 推出基于区块链的证券发行平台，号称每秒达到 10 万笔交易。
- DAH 为金融市场交易提供基于区块链的交易系统。获得澳洲证交所项目。
- Symbiont 帮助金融企业创建存储于区块链的智能债券，当条件符合时，清算立即执行。
- Overstock.com 推出基于区块链的私有和公开股权交易“T0”平台，提出“交易即结算”(The trade is the settlement)的理念，主要目标是建立证券交易实时清算结算的全新系统。
- 高盛为一种叫做“SETLcoin”的新虚拟货币申请专利，用于为股票和债券等资产交易提供“近乎立即执行和结算”的服务。

征信和权属管理

征信管理

征信管理是一个巨大的潜在市场，据称超过千亿规模（平安证券报告，美国富国银行报告），也是目前大数据应用最有前途的方向之一。

目前的征信相关的大量有效数据主要集中在少数机构手中。由于这些数据太过敏感，并且是商业命脉，往往会被严密保护起来，进而形成很高的行业门槛。

虽然现在大量的互联网企业（最成功的应该属 facebook）尝试从各种维度都获取了海量的用户信息，但从征信角度看，这些数据仍然存在若干问题。

- 数据量不足：数据量越大，能获得的价值自然越高，而数据产生有效价值存在一个下限。低于下限的数据量无法产生有效价值；
- 相关度较差：最核心的数据也往往是最敏感的，在隐私高度敏感的今天，用户都不希望暴露过多数据给第三方，因此企业获取到数据中有效成分其实很少；
- 时效性不足：企业可以从明面上获取到的用户数据往往是过时的，甚至存在虚假信息，对相关分析的可信度造成严重干扰。

而区块链存在着天然无法篡改、不可抵赖的特性。同时，区块链将可能提供前所未有规模的相关性极高的数据，这些数据可以在时空中准确定位，并严格关联到用户。因此，基于区块链提供数据进行征信管理，将让信用评估的准确率大大提高，并且降低进行评估的成本。

另外，跟传统依靠人工的审核不同，区块链技术完全依靠数学成果，基于区块链的信用机制将天然具备稳定性和中立性。

包括 IDG、腾讯、安永、普华永道等都纷纷投资或进入基于区块链的征信管理领域，特别是跟保险和互助经济相关的应用场景。

权属管理

用于产权、版权等所有权管理和追踪。包括汽车、房屋、艺术品等各种贵重物品的交易等。也包括数字出版物，以及可以标记实体物品的数字标记。

目前最大的几个难题是：

- 物品所有权的确认和管理；
- 交易的安全可靠；
- 一定的隐私保护。

比如，目前要交易房屋，如果买卖双方互相不认识的话，往往需要依托中介机构来确保交易的进行，通过纸质的材料证明房屋所有权。但实际上，很多时候中介机构也无法确保交易的正常进行。

而利用区块链技术，物品的所有权是写在数字链上的，谁都无法修改，并且一旦出现合同中约定情况，区块链技术将确保合同能得到准确执行。

公正通（Factom）尝试使用区块链技术来革新商业社会和政府部门的数据管理和数据记录方式。包括审计系统、医疗信息记录、供应链管理、投票系统、财产契据、法律应用、金融系统等。它将待确权数据的指纹存放到基于区块链的分布式账本中，可以提供资产所有权的追踪服务。

其它项目

在教育领域，MIT 研究员朱莉安娜·纳扎雷（Juliana Nazaré）和学术创新部主管菲利普·施密特（Philipp Schmidt）发表了 文章，介绍基于区块链的学历认证系统。基于该系统，用人单位可以确认求职者的学历信息是真实可靠的。

此外，还包括：

- BitShare：自由贸易的资产交易所。
- Everledger：基于区块链的贵重资产检测系统，将钻石或者艺术品加上哈希值记录在区块链上。
- Mycelia：区块链产权保护项目，为音乐人实现音乐的自由交易。
- Monegraph：通过区块链保障图片版权的透明交易。
- Mediachain：通过 metadata 协议，将内容创造者与作品唯一对应。

资源共享

资源共享目前面临的问题主要包括：

- 共享过程成本过高；
- 用户身份评分难
- 共享服务管理难

短租共享

大量提供短租服务的公司已经开始尝试用区块链来解决共享中的难题。

一份来着 高盛的报告 中宣称：

> Airbnb 等 P2P 住宿平台已经开始通过利用私人住所打造公开市场来变革住宿行业，但是这种服务的接受程度可能会因人们对人身安全以及财产损失的担忧而受到限制。而如果通过引入安全且无法篡改的数字化资质和信用管理系统，我们认为区块链就能有助于提升P2P住宿的接受度。

该报告还指出，可能采用区块链技术的企业 Airbnb、HomeAway 以及 OneFineStay 等，市场规模为 30-90 亿美元。

社区能源共享

案例主要包括家庭太阳能发电后通过社区的电力网络进行买卖，例如纽约的 微型电网。

ConsenSys 和微电网开发商 LO3 共建光伏发电交易网络，实现点对点的能源交易。

主要难题包括：

- 太阳能电池
- 社区电网构建
- 电力储备系统
- 交易系统

现在已经有大量创业团队在解决这些问题，硬件部分已经有了很多很好的案例。而通过区块链技术打造的平台主要解决最后一个问题，可以很容易实现社区内低成本的可靠交易系统。

电商平台

OpenBazaar 试图在无中介的情形下，实现安全电商交易。

传统情况下，电商平台起到了中介的作用，一旦发生纠纷，会作为第三方机构进行审判。这种模式存在着周期长、缺乏公证、成本高等缺点。

OpenBazaar 通过多方签名机制和信誉评分机制，让众多参与者合作进行评估，零成本解决纠纷问题。

大数据共享

大数据时代里，价值来自于对数据的挖掘，数据维度越多，体积越大，潜在价值也就越高。

一直以来，比较让人头疼的问题是如何评估数据的价值，如何利用数据进行交换和交易，以及如何避免宝贵的数据在未经许可的情况下泄露出去。

区块链技术为解决这些问题提供了潜在的可能。

利用区块链构成的统一账本，数据在多方之间的流动将得到实时地追踪和管理，并且通过对访问权限的管控，可以有效降低对数据共享过程的管理成本。

投资管理

跨境贸易

在国际贸易活动中，买卖双方可能互不信任。因此需要两家银行作为买卖双方的保证人，代为收款交单，并以银行信用代替商业信用。

区块链可以为信用证交易参与方提供共同账本，允许银行和其它参与方拥有经过确认的共同交易记录并据此履约，从而降低风险和成本。

一带一路

一带一路中对区块链技术的探索应用，能让原先无法交易的双方（例如，不存在都认可的国际货币情况下）完成交易，并且降低贸易风险、减少成本。

众筹投资

以 DAO（Decentralized Autonomous Organization）为代表的众筹管理，DAO 曾创下历史最高的融资记录，超过 1.6 亿美金。

物联网

曾经有人认为，物联网为大数据时代的基础。

笔者认为，区块链技术是物联网时代的基础。

应用场景分析

一种可能的应用场景为：通过 `Transaction` 产生对应的行为，为每一个设备分配地址 `Address`，给该地址注入一定的费用，可以执行相关动作，从而达到物联网的应用。类似于：`PM2.5监测点` 数据获取，`服务器` 租赁，`网络摄像头` 数据调用，`DNS服务器` 等。

另外，随着物联网设备的增多，Edge 计算需求的增强，大量设备之间需要通过分布式自组织的管理模式，并且对容错性要求很高。区块链自身分布式和抗攻击的特点可以很好地试用到这一场景中。

IBM

IBM 在物联网领域已经持续投入了几十年的研发，目前正在探索使用区块链技术来降低物联网应用的成本。

2015 年初，IBM 与三星宣布合作研发 ADEPT 系统。

物流供应链

供应链行业往往涉及到诸多实体，包括物流、资金流、信息流等，这些实体之间存在大量复杂的协作和沟通。传统模式下，不同实体各自保存各自的供应链信息，严重缺乏透明度，造成了较高的时间成本和金钱成本，而且一旦出现问题（冒领、货物假冒等）难以追查和处理。

通过区块链各方可以获得一个透明可靠的统一信息平台，可以实时查看状态，降低物流成本，追溯物品的生产和运送整个过程，从而提高供应链管理的效率。当发生纠纷时，举证和追查也变得更加清晰和容易。

该领域被认为是区块链一个很有前景的应用方向。

例如运送方通过扫描二维码来证明货物到达指定区域，并自动收取提前约定的费用，可以参考 区块链如何变革供应链金融 和 区块链给供应链带来透明。

Skuchain 创建基于区块链的新型供应链解决方案，实现商品流与资金流的同步，同时缓解假货问题。

公共网络服务

现有的互联网能正常运行，离不开很多近乎免费的网络服务，例如域名服务（DNS）。任何人都可以免费查询到域名，没有 DNS，现在的各种网站基本就无法访问了。因此，对于网络系统来说，类似的基础服务必须要能做到安全可靠，并且低成本。

区块链技术恰好具备这些特点，基于区块链打造的 DNS 系统，将不再会出现各种错误的查询结果，并且可以稳定可靠的提供服务。

其它场景

还有一些很有趣的应用场景。主要包括：

- BitMessage：基于区块链的安全可靠的通信系统。
- GemHealth：医疗数据的安全管理，已与医疗行业多家公司签订了合作协议。
- Storj：基于比特币区块链的安全的数据分布式存储服务。
- Tierion：确保数据安全记录。
- Twister：去中心化的“微博”系统。

小结

本章介绍了大量的区块链技术应用案例和未来场景，讨论了区块链作为一项基础技术，所具有的市场潜力。

当然，任何事物的发展都不是一帆风顺的。

目前来看，制约区块链技术进一步应用的因素有很多。首先就是谁来为区块链上的合同担保？特别在金融、法律等领域，实际执行的生活往往还得是由人来做；另外就是物品的数字化。非数字化的物品很难放到数字世界中进行管理。

这些问题都不是很容易就得到解决的，但笔者相信，看一个东西成不成，根子上还是看它有没有提高生产力。随着众多行业对区块链技术的试水和探索，一定会有更多的应用场景出现。

分布式系统

万法皆空，因果不空。

随着摩尔定律碰到瓶颈，越来越多的系统要依靠分布式集群架构来实现海量数据处理和可扩展计算能力。

区块链首先是一个分布式系统。

中央式结构改成分布式系统，碰到的第一个问题就是一致性的保障。

很显然，如果一个分布式集群无法保证处理结果一致的话，那任何建立于其上的业务系统都无法正常工作。

本章将介绍分布式系统中一些核心问题的来源以及相关的工作。

一致性问题

在分布式系统中，一致性(Consistency，早期也叫 Agreement)是指对于系统中的多个服务节点，给定一系列操作，在协议（往往通过某种共识算法）保障下，试图使得它们对处理结果达成某种程度的一致。

如果分布式系统能实现“一致”，对外就可以呈现为一个功能正常的，且性能和稳定性都要好很多的“虚处理节点”。

举个例子，某影视公司旗下有西单和中关村的两个电影院，都出售某电影票，票一共就一万张。那么，顾客到达某个电影院买票的时候，售票员该怎么决策是否该卖这张票，才能避免超售呢？当电影院个数更多的时候呢？

这个问题在人类世界中，看起来似乎没那么难，你看，英国人不是刚靠 投票 达成了“某种一致”吗？

注意：一致性并不代表结果正确与否，而是系统对外呈现的状态一致与否，例如，所有节点都达成失败状态也是一种一致。

挑战

在实际的计算机集群系统（看似强大的计算机系统，很多地方都比人类世界要脆弱的多）中，存在如下的问题：

- 节点之间的网络通讯是不可靠的，包括任意延迟和内容故障；
- 节点的处理可能是错误的，甚至节点自身随时可能宕机；
- 同步调用会让系统变得不具备可扩展性。

要解决这些挑战，愿意动脑筋的读者可能会很快想出一些不错的思路。

为了简化理解，仍然以两个电影院一起卖票的例子。可能有如下的解决思路：

- 每次要卖一张票前打电话给另外一家电影院，确认下当前票数并没超售；
- 两家电影院提前约好，奇数小时内一家可以卖票，偶数小时内另外一家可以卖；
- 成立一个第三方的存票机构，票都放到他那里，每次卖票找他询问；
- 更多……

这些思路大致都是可行的。实际上，这些方法背后的思想，将可能引发不一致的并行操作进行串行化，就是现在计算机系统里处理分布式一致性问题的基础思路和唯一秘诀。只是因为计算机系统比较傻，需要考虑得更全面一些；而人们又希望计算机系统能工作的更快更稳定，所以算法需要设计得再精巧一些。

要求

规范的说，理想的分布式系统一致性应该满足：

- 可终止性（Termination）：一致的结果在有限时间内能完成；
- 共识性（Consensus）：不同节点最终完成决策的结果应该相同；
- 合法性（Validity）：决策的结果必须是其它进程提出的提案。

第一点很容易理解，这是计算机系统可以被使用的前提。需要注意，在现实生活中这点并不是总能得到保障的，例如取款机有时候会是“服务中断”状态，电话有时候是“无法连通”的。

第二点看似容易，但是隐藏了一些潜在信息。算法考虑的是任意的情形，凡事一旦推广到任意情形，就往往有一些惊人的结果。例如现在就剩一张票了，中关村和西单的电影院也分别刚确认过这张票的存在，然后两个电影院同时来了一个顾客要买票，从各自“观察”看来，自己的顾客都是第一个到的……怎么能达成结果的共识呢？记住我们的唯一秘诀：核心在于需要把两件事情进行排序，而且这个顺序还得是大家都认可的。

第三点看似绕口，但是其实比较容易理解，即达成的结果必须是节点执行操作的结果。仍以卖票为例，如果两个影院各自卖出去一千张，那么达成的结果就是还剩八千张，决不能认为票售光了。

带约束的一致性

做过分布式系统的读者应该能意识到，绝对理想的强一致性（Strong Consistency）代价很大。除非不发生任何故障，所有节点之间的通信无需任何时间，这个时候其实就等价于一台机器了。实际上，越强的一致性要求往往意味着越弱的性能。

一般的，强一致性（Strong Consistency）主要包括下面两类：

- 顺序一致性（Sequential Consistency）：Leslie Lamport 1979 年经典论文《How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs》中提出，是一种比较强的约束，保证所有进程看到的 全局执行顺序（total order）一致，并且每个进程看自身的执行（local order）跟实际发生顺序一致。例如，某进程先执行 A，后执行 B，则实际得到的全局结果中就应该为 A 在 B 前面，而不能反过来。同时所有其它进程在全局上也应该看到这个顺序。顺序一致性实际上限制了各进程内指令的偏序关系，但不在进程间按照物理时间进行全局排序。
- 线性一致性（Linearizability Consistency）：Maurice P. Herlihy 与 Jeannette M. Wing 在 1990 年经典论文《Linearizability: A Correctness Condition for Concurrent Objects》中共同提出，在顺序一致性前提下加强了进程间的操作排序，形成唯一的全局顺序（系统等价于是顺序执行，所有进程看到的所有操作的序列顺序都一致，并且跟实际发生顺序一致），是很强的原子性保证。但是比较难实现，目前基本上要么依赖于全局的时钟或锁，要么通过一些复杂算法实现，性能往往不高。

目前，高精度的石英钟的漂移率为 10^{-7}，人类目前最准确的原子震荡时钟的漂移率为 10^{-13}。Google 曾在其分布式数据库 Spanner 中采用基于原子时钟和 GPS 的“TrueTime”方案，能够将不同数据中心的时间偏差控制在 10ms 以内。方案简单粗暴而有效，但存在成本较高的

问题。

强一致的系统往往比较难实现。很多时候，人们发现实际需求并没有那么强，可以适当放宽一致性要求，降低系统实现的难度。例如在一定约束下实现所谓最终一致性（Eventual Consistency），即总会存在一个时刻（而不是立刻），系统达到一致的状态，这对于大部分的 Web 系统来说已经足够了。这一类弱化的一致性，被笼统称为弱一致性（Weak Consistency）。

莫非分布式领域也有一个测不准原理？这个世界为何会有这么多的约束呢？

共识算法

实际上，要保障系统满足不同程度的一致性，往往需要通过共识算法来达成。

共识算法解决的是对某个提案（Proposal），大家达成一致意见的过程。提案的含义在分布式系统中十分宽泛，如多个事件发生的顺序、某个键对应的值、谁是领导……等等，可以认为任何需要达成一致的信息都是一个提案。

注：实践中，一致性的结果往往还需要客户端的特殊支持，典型地通过访问足够多个服务节点来验证确保获取共识后结果。

问题挑战

实际上，如果分布式系统中各个节点都能保证以十分强大的性能（瞬间响应、高吞吐）无故障的运行，则实现共识过程并不复杂，简单通过多播过程投票即可。

很可惜的是，现实中这样“完美”的系统并不存在，如响应请求往往存在时延、网络会发生中断、节点会发生故障、甚至存在恶意节点故意要破坏系统。

一般地，把故障（不响应）的情况称为“非拜占庭错误”，恶意响应的情况称为“拜占庭错误”（对应节点为拜占庭节点）。

常见算法

针对非拜占庭错误的情况，一般包括 Paxos、Raft 及其变种。

对于要能容忍拜占庭错误的情况，一般包括 PBFT 系列、PoW 系列算法等。从概率角度，PBFT 系列算法是确定的，一旦达成共识就不可逆转；而 PoW 系列算法则是不确定的，随着时间推移，被推翻的概率越来越小。

理论界限

搞学术的人都喜欢对问题先确定一个界限，那么，这个问题的最坏界限在哪里呢？很不幸，一般情况下，分布式系统的共识问题无解。

当节点之间的通信网络自身不可靠情况下，很显然，无法确保实现共识。但好在，一个设计得当的网络可以在大概率上实现可靠的通信。

然而，即便在网络通信可靠情况下，一个可扩展的分布式系统的共识问题的下限是无解。

这个结论，被称为 `FLP 不可能性` 原理，可以看做分布式领域的“测不准原理”。

FLP 不可能性原理

> FLP 不可能原理：在网络可靠，存在节点失效（即便只有一个）的最小化异步模型系统中，不存在一个可以解决一致性问题的确定性算法。

提出该定理的论文是由 Fischer, Lynch 和 Patterson 三位作者于 1985 年发表，该论文后来获得了 Dijkstra（就是发明最短路径算法的那位）奖。

FLP 不可能原理实际上告诉人们，不要浪费时间去为异步分布式系统设计在任意场景下都能实现共识的算法。

理解这一原理的一个不严谨的例子是：

三个人在不同房间，进行投票（投票结果是 0 或者 1）。三个人彼此可以通过电话进行沟通，但经常会有人时不时地睡着。比如某个时候，A 投票 0，B 投票 1，C 收到了两人的投票，然后 C 睡着了。A 和 B 则永远无法在有限时间内获知最终的结果。如果可以重新投票，则类似情形每次在取得结果前发生：（

FLP 原理实际上说明对于允许节点失效情况下，纯粹异步系统无法确保一致性在有限时间内完成。

这岂不是意味着研究一致性问题压根没有意义吗？

先别这么悲观，学术界做研究，考虑的是数学和物理意义上最极端的情形，很多时候现实生活要美好的多（感谢这个世界如此鲁棒！）。例如，上面例子中描述的最坏情形，总会发生的概率并没有那么大。工程实现上多试几次，很大可能就成功了。

科学告诉你什么是不可能的；工程则告诉你，付出一些代价，我可以把它变成可能。

这就是工程的魅力。

那么，退一步讲，在付出一些代价的情况下，我们能做到多少？

回答这一问题的是另一个很出名的原理：CAP 原理。

科学上告诉你去赌场赌博从概率上总会是输钱的；工程则告诉你，如果你愿意接受最终输钱的结果，中间说不定偶尔能小赢几笔呢！？

CAP 原理

CAP 原理最早由 Eric Brewer 在 2000 年，ACM 组织的一个研讨会上提出猜想，后来 Lynch 等人进行了证明。

该原理被认为是分布式系统领域的重要原理。

定义

分布式计算系统不可能同时确保一致性（Consistency）、可用性（Availablity）和分区容忍性（Partition），设计中往往需要弱化对某个特性的保证。

- 一致性（Consistency）：任何操作应该都是原子的，发生在后面的事件能看到前面事件发生导致的结果，注意这里指的是强一致性；
- 可用性（Availablity）：在有限时间内，任何非失败节点都能应答请求；
- 分区容忍性（Partition）：网络可能发生分区，即节点之间的通信不可保障。

比较直观地理解，当网络可能出现分区时候，系统是无法同时保证一致性和可用性的。要么，节点收到请求后因为没有得到其他人的确认就不应答，要么节点只能应答非一致的结果。

好在大部分时候网络被认为是可靠的，因此系统可以提供一致可靠的服务；当网络不可靠时，系统要么牺牲掉一致性（大部分时候都是如此），要么牺牲掉可用性。

应用场景

既然 CAP 不可同时满足，则设计系统时候必然要弱化对某个特性的支持。

弱化一致性

对结果一致性不敏感的应用，可以允许在新版本上线后过一段时间才更新成功，期间不保证一致性。

例如网站静态页面内容、实时性较弱的查询类数据库等，CouchDB、Cassandra 等为此设计。

弱化可用性

对结果一致性很敏感的应用，例如银行取款机，当系统故障时候会拒绝服务。MongoDB、Redis 等为此设计。

Paxos、Raft 等算法，主要处理这种情况。

弱化分区容忍性

现实中，网络分区出现概率减小，但较难避免。某些关系型数据库、ZooKeeper 即为此设计。

实践中，网络通过双通道等机制增强可靠性，达到高稳定的网络通信。

ACID 原则

即 Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）、Durability（持久性）。

ACID 原则描述了对分布式数据库的一致性需求，同时付出了可用性的代价。

- Atomicity：每次操作是原子的，要么成功，要么不执行；
- Consistency：数据库的状态是一致的，无中间状态；
- Isolation：各种操作彼此互相不影响；
- Durability：状态的改变是持久的，不会失效。

一个与之相对的原则是 BASE（Basic Availiability，Soft state，Eventually Consistency），牺牲掉对一致性的约束（最终一致性），来换取一定的可用性。

Paxos 与 Raft

Paxos 问题是指分布式的系统中存在故障（fault），但不存在恶意（corrupt）节点场景（即可能消息丢失或重复，但无错误消息）下的共识达成（Consensus）问题。因为最早是 Leslie Lamport 用 Paxon 岛的故事模型来进行描述而命名。

Paxos

1990 年由 Leslie Lamport 提出的 Paxos 共识算法，在工程角度实现了一种最大化保障分布式系统一致性（存在极小的概率无法实现一致）的机制。Paxos 被广泛应用在 Chubby、ZooKeeper 这样的系统中，Leslie Lamport 因此获得了 2013 年度图灵奖。

故事背景是古希腊 Paxon 岛上的多个法官在一个大厅内对一个议案进行表决，如何达成统一的结果。他们之间通过服务人员来传递纸条，但法官可能离开或进入大厅，服务人员可能偷懒去睡觉。

Paxos 是第一个被证明的共识算法，其原理基于 两阶段提交 并进行扩展。

作为现在共识算法设计的鼻祖，以最初论文的难懂（算法本身并不复杂）出名。算法中将节点分为三种类型：

- proposer：提出一个提案，等待大家批准为结案。往往是客户端担任该角色；
- acceptor：负责对提案进行投票。往往是服务端担任该角色；
- learner：被告知结案结果，并与之统一，不参与投票过程。可能为客户端或服务端。

并且，算法需要满足 safety 和 liveness 两方面的约束要求（实际上这两个基础属性是大部分分布式算法都该考虑的）：

- safety：保证决议结果是对的，无歧义的，不会出现错误情况。
  - 决议（value）只有在被 proposers 提出的 proposal 才能被最终批准；
  - 在一次执行实例中，只批准（chosen）一个最终决议，意味着多数接受（accept）的结果能成为决议；
- liveness：保证决议过程能在有限时间内完成。
  - 决议总会产生，并且 learners 能获得被批准（chosen）的决议。

基本过程包括 proposer 提出提案，先争取大多数 acceptor 的支持，超过一半支持时，则发送结案结果给所有人进行确认。一个潜在的问题是 proposer 在此过程中出现故障，可以通过超时机制来解决。极为凑巧的情况下，每次新的一轮提案的 proposer 都恰好故障，系统则永远无法达成一致（概率很小）。

Paxos 能保证在超过 $1/2$ 的正常节点存在时，系统能达成共识。

读者可以试着自己设计一套能达成共识的方案，会发现在满足各种约束情况下，算法自然就会那样设计。

单个提案者+多接收者

如果系统中限定只有某个特定节点是提案者，那么一致性肯定能达成（只有一个方案，要么达成，要么失败）。提案者只要收到了来自多数接收者的投票，即可认为通过，因为系统中不存在其他的提案。

但一旦提案者故障，则系统无法工作。

多个提案者+单个接收者

限定某个节点作为接收者。这种情况下，共识也很容易达成，接收者收到多个提案，选第一个提案作为决议，拒绝掉后续的提案即可。

缺陷也是容易发生单点故障，包括接收者故障或首个提案者节点故障。

以上两种情形其实类似主从模式，虽然不那么可靠，但因为原理简单而被广泛采用。

当提案者和接收者都推广到多个的情形，会出现一些挑战。

多个提案者+多个接收者

既然限定单提案者或单接收者都会出现故障，那么就得允许出现多个提案者和多个接收者。问题一下子变得复杂了。

一种情况是同一时间片段（如一个提案周期）内只有一个提案者，这时可以退化到单提案者的情形。需要设计一种机制来保障提案者的正确产生，例如按照时间、序列、或者大家猜拳（出一个数字来比较）之类。考虑到分布式系统要处理的工作量很大，这个过程要尽量高效，满足这一条件的机制非常难设计。

另一种情况是允许同一时间片段内可以出现多个提案者。那同一个节点可能收到多份提案，怎么对他们进行区分呢？这个时候采用只接受第一个提案而拒绝后续提案的方法也不适用。很自然的，提案需要带上不同的序号。节点需要根据提案序号来判断接受哪个。比如接受其中序号较大（往往意味着是接受新提出的，因为旧提案者故障概率更大）的提案。

如何为提案分配序号呢？一种可能方案是每个节点的提案数字区间彼此隔离开，互相不冲突。为了满足递增的需求可以配合用时间戳作为前缀字段。

此外，提案者即便收到了多数接收者的投票，也不敢说就一定通过。因为在此过程中系统可能还有其它的提案。

两阶段的提交

提案者发出提案之后，收到一些反馈。一种结果是自己的提案被大多数接受了，另一种结果是没被接受。没被接受的话好说，过会再试试。

即便收到来自大多数的接受反馈，也不能认为就最终确认了。因为这些接收者自己并不知道自己刚反馈的提案就恰好是全局的绝大多数。

很自然的，引入了新的一个阶段，即提案者在前一阶段拿到所有的反馈后，判断这个提案是可能被大多数接受的提案，需要对其进行最终确认。

Paxos 里面对这两个阶段分别命名为准备（prepare）和提交（commit）。准备阶段解决大家对哪个提案进行投票的问题，提交阶段解决确认最终值的问题。

准备阶段：

- 提案者发送自己计划提交的提案的编号到多个接受者，试探是否可以锁定多数接受者的支持。
- 接受者时刻保留收到过提案的最大编号和接受的最大提案。如果收到的提案号比目前保留的最大提案号还大，则返回自己已接受的提案值（如果还未接受过任何提案，则为空）给提案者，更新当前最大提案号，并说明不再接受小于最大提案号的提案。

提交阶段：

- 提案者如果收到大多数的回复（表示大部分人听到它的请求），则可准备发出带有刚才提案号的接受消息。如果收到的回复中不带有新的提案，说明锁定成功，则使用自己的提案内容；如果返回中有提案内容，则替换提案值为返回中编号最大的提案值。如果没收到足够多的回复，则需要再次发出请求。
- 接受者收到接受消息后，如果发现提案号不小于已接受的最大提案号，则接受该提案，并更新接受的最大提案。

一旦多数接受了共同的提案值，则形成决议，成为最终确认的提案。

Raft

Raft 算法是Paxos 算法的一种简化实现。

包括三种角色：leader、candidate 和 follower，其基本过程为：

- Leader 选举：每个 candidate 随机经过一定时间都会提出选举方案，最近阶段中得票最多者被选为 leader；
- 同步 log：leader 会找到系统中 log 最新的记录，并强制所有的 follower 来刷新到这个记录；

注：此处 *log* 并非是指日志消息，而是各种事件的发生记录。

拜占庭问题与算法

拜占庭问题更为广泛，讨论的是允许存在少数节点作恶（消息可能被伪造）场景下的一致性达成问题。拜占庭算法讨论的是最坏情况下的保障。

中国将军问题

拜占庭将军问题之前，就已经存在中国将军问题：两个将军要通过信使来达成进攻还是撤退的约定，但信使可能迷路或被敌军阻拦（消息丢失或伪造），如何达成一致。根据 FLP 不可能原理，这个问题无解。

拜占庭问题

又叫拜占庭将军（Byzantine Generals Problem）问题，是 Leslie Lamport 1982 年提出用来解释一致性问题的一个虚构模型。拜占庭是古代东罗马帝国的首都，由于地域宽广，守卫边境的多个将军（系统中的多个节点）需要通过信使来传递消息，达成某些一致的决定。但由于将军中可能存在叛徒（系统中节点出错），这些叛徒将努力向不同的将军发送不同的消息，试图会干扰一致性的达成。

拜占庭问题即为在此情况下，如何让忠诚的将军们能达成行动的一致。

对于拜占庭问题来说，假如节点总数为 N，叛变将军数为 F，则当 时，问题才有解，即 Byzantine Fault Tolerant (BFT) 算法。

例如，$N=3,F=1$ 时。

提案人不是叛变者，提案人发送一个提案出来，叛变者可以宣称收到的是相反的命令。则对于第三个人（忠诚者）收到两个相反的消息，无法判断谁是叛变者，则系统无法达到一致。

提案人是叛变者，发送两个相反的提案分别给另外两人，另外两人都收到两个相反的消息，无法判断究竟谁是叛变者，则系统无法达到一致。

更一般的，当提案人不是叛变者，提案人提出提案信息 1，则对于合作者来看，系统中会有 $N-F$ 份确定的信息 1，和 F 份不确定的信息（可能为 0 或 1，假设叛变者会尽量干扰一致的达成），$N-F>F$，即 $N>2F$ 情况下才能达成一致。

当提案人是叛变者，会尽量发送相反的提案给 $N-F$ 个合作者，从收到 1 的合作者看来，系统中会存在 个信息 1，以及 个信息 0 ；从收到 0 的合作者看来，系统中会存在 个信息 0 ，以及 个信息 1 ；

另外存在 $F-1$ 个不确定的信息。合作者要想达成一致，必须进一步的对所获得的消息进行判定，询问其他人某个被怀疑对象的消息值，并通过取多数来作为被怀疑者的信息值。这个过程可以进一步递归下去。

Leslie Lamport 证明，当叛变者不超过 时，存在有效的算法，不论叛变者如何折腾，忠诚的将军们总能达成一致的结果。如果叛变者过多，则无法保证一定能达到一致性。

多于 的叛变者时有没有可能有解决方案呢？设想 f 个叛变者和 g 个忠诚者，叛变者故意使坏，可以给出错误的结果，也可以不响应。某个时候 f 个叛变者都不响应，则 g 个忠诚者取多数既能得到正确结果。当 f 个叛变者都给出一个恶意的提案，并且 g 个忠诚者中有 f 个离线时，剩下的 $g-f$ 个忠诚者此时无法分别是否混入了叛变者，仍然要确保取多数能得到正确结果，因此，$g-f>f$，即 $g>2f$，所以系统整体规模要大于 $3f$。

能确保达成一致的拜占庭系统节点数至少为 4，允许出现 1 个坏的节点。

Byzantine Fault Tolerant 算法

面向拜占庭问题的容错算法，解决的是网络通信可靠，但节点可能故障情况下的一致性达成。

最早由 Castro 和 Liskov 在 1999 年提出的 Practical Byzantine Fault Tolerant（PBFT）是第一个得到广泛应用的 BFT 算法。只要系统中有 的节点是正常工作的，则可以保证一致性。

PBFT 算法包括三个阶段来达成共识：Pre-Prepare、Prepare 和 Commit。

新的解决思路

拜占庭问题之所以难解，在于任何时候系统中都可能存在多个提案（因为提案成本很低），并且要完成最终的一致性确认过程十分困难，容易受干扰。但是一旦确认，即为最终确认。

比特币的区块链网络在设计时提出了创新的 PoW（Proof of Work） 算法思路。一个是限制一段时间内整个网络中出现提案的个数（增加提案成本），另外一个是放宽对最终一致性确认的需求，约定好大家都确认并沿着已知最长的链进行拓宽。系统的最终确认是概率意义上的存在。这样，即便有人试图恶意破坏，也会付出很大的经济代价（付出超过系统一半的算力）。

后来的各种 PoX 系列算法，也都是沿着这个思路进行改进，采用经济上的惩罚来制约破坏者。

可靠性指标

很多领域一般都喜欢谈服务可靠性，用几个 9 来说事。这几个 9 其实是粗略代表了概率意义上系统能提供服务的可靠性指标，最初是电信领域提出的概念。

下表给出不同指标下，每年允许服务出现不可用时间的参考值。

指标	概率可靠性	每年允许不可用时间	典型场景
一个九	90%	1.2 个月	不可用
二个九	99%	3.6 天	普通单点
三个九	99.9%	8.6 小时	普通企业
四个九	99.99%	51.6 分钟	高可用
五个九	99.999%	5 分钟	电信级
六个九	99.9999%	31 秒	极高要求
七个九	99.99999%	3 秒	N/A
八个九	99.999999%	0.3 秒	N/A
九个九	99.9999999%	30 毫秒	N/A

一般来说，单点的服务器系统至少应能满足两个九；普通企业信息系统三个九就肯定足够了（大家可以统计下自己企业内因系统维护每年要停多少时间），系统能达到四个九已经是业界领先水平了（参考 AWS）。电信级的应用一般号称能达到五个九，这已经很厉害了，一年里面最多允许五分钟的服务停用。六个九和以上的系统，就更加少见了，要实现往往意味着极高的代价。

那么，该如何提升可靠性呢？有两个思路：一是让系统中的单点变得更可靠；二是消灭单点。

IT 从业人员大都有类似的经验，运行某软系统的机器，基本上是过几天就要重启下的；而运行 Linux 系统的服务器，则可能几年时间都不出问题。另外，普通的家用计算机，跟专用服务器相比，长时间运行更容易出现故障。这些都是单点可靠性不同的例子。可以通过替换单点的软硬件来改善可靠性。

然而，依靠单点实现的可靠性毕竟是有限的，要想进一步的提升，那就只好消灭单点，通过主从、多活等模式让多个节点集体完成原先单点的工作。这可以从概率意义上改善服务的可靠性，也是分布式系统的一个重要用途。

小结

分布式系统领域是计算机科学中十分重要的一个技术领域。

常见的分布式一致性是个古老而重要的问题，无论在学术上还是工程上都存在很高的价值。理想化（各项指标均最优）的解决方案是不存在的。

在现实各种约束条件下，往往需要通过牺牲掉某些需求，来设计出满足特定场景的协议。

其实，工程领域中很多问题的解决思路，都在于如何合理地进行取舍（trade-off）。

密码学技术

工程领域从来没有黑科技；密码学不是工程。

密码学在信息技术领域的重要地位无需多言。如果没有现代密码学的研究成果，人类社会根本无法进入信息时代。

密码学领域十分繁杂，本章将介绍密码学领域中跟区块链相关的一些基础知识，包括 hash 算法与摘要、加密算法、数字签名和证书、PKI 体系、Merkle 树、同态加密等，以及如何使用这些技术实现信息的机密性、完整性、认证性和不可抵赖性。

Hash 算法

定义

Hash （哈希或散列）算法是信息技术领域非常基础也非常重要的技术。它能任意长度的二进制值（明文）映射为较短的固定长度的二进制值（Hash 值），并且不同的明文很难映射为相同的 Hash 值。

例如计算一段话“hello blockchain world, this is yeasy@github”的 MD5 hash 值为 `89242549883a2ef85dc81b90fb606046` 。

```
$ echo "hello blockchain world, this is yeasy@github"|md5
89242549883a2ef85dc81b90fb606046
```

这意味着我们只要对某文件进行 MD5 Hash 计算，得到结果为 `89242549883a2ef85dc81b90fb606046` ，这就说明文件内容极大概率上就是 “hello blockchain world, this is yeasy@github”。可见，Hash 的核心思想十分类似于基于内容的编址或命名。

注：*hash* 值在应用中又被称为指纹（*fingerprint*）、摘要（*digest*）。

注：*MD5* 是一个经典的 *hash* 算法，其和 *SHA-1* 算法都已被 证明 安全性不足应用于商业场景。

一个优秀的 hash 算法，将能实现：

- 正向快速：给定明文和 hash 算法，在有限时间和有限资源内能计算出 hash 值。
- 逆向困难：给定（若干） hash 值，在有限时间内很难（基本不可能）逆推出明文。
- 输入敏感：原始输入信息修改一点信息，产生的 hash 值看起来应该都有很大不同。
- 冲突避免：很难找到两段内容不同的明文，使得它们的 hash 值一致（发生冲突）。

冲突避免有时候又被称为“抗碰撞性”。如果给定一个明文前提下，无法找到碰撞的另一个明文，称为“弱抗碰撞性”；如果无法找到任意两个明文，发生碰撞，则称算法具有“强抗碰撞性”。

很多场景下，也要求对于任意长的输入内容，输出定长的 *hash* 结果。

流行的算法

目前流行的 Hash 算法包括 MD5、SHA-1 和 SHA-2。

MD4（RFC 1320）是 MIT 的 Ronald L. Rivest 在 1990 年设计的，MD 是 Message Digest 的缩写。其输出为 128 位。MD4 已证明不够安全。

MD5（RFC 1321）是 Rivest 于1991年对 MD4 的改进版本。它对输入仍以 512 位分组，其输出是 128 位。MD5 比 MD4 复杂，并且计算速度要慢一点，更安全一些。MD5 已被证明不具备“强抗碰撞性”。

SHA（Secure Hash Algorithm）是一个 Hash 函数族，由 NIST（National Institute of Standards and Technology）于 1993 年发布第一个算法。目前知名的 SHA-1 在 1995 年面世，它的输出为长度 160 位的 hash 值，因此抗穷举性更好。SHA-1 设计时基于和 MD4 相同原理，并且模仿了该算法。SHA-1 已被证明不具备“强抗碰撞性”。

为了提高安全性，NIST 还设计出了 SHA-224、SHA-256、SHA-384，和 SHA-512 算法（统称为 SHA-2），跟 SHA-1 算法原理类似。SHA-3 相关算法也已被提出。

目前，一般认为 MD5 和 SHA1 已经不够安全，推荐至少使用 SHA2-256 算法。

性能

一般的，Hash 算法都是算力敏感型，意味着计算资源是瓶颈，主频越高的 CPU 进行 Hash 的速度也越快。

也有一些 Hash 算法不是算力敏感的，例如 scrypt，需要大量的内存资源，节点不能通过简单的增加更多 CPU 来获得 hash 性能的提升。

数字摘要

顾名思义，数字摘要是对数字内容进行 Hash 运算，获取唯一的摘要值来指代原始数字内容。

数字摘要是解决确保内容没被篡改过的问题（利用 Hash 函数的抗碰撞性特点）。

数字摘要是 Hash 算法最重要的一个用途。在网络上下载软件或文件时，往往同时会提供一个数字摘要值，用户下载下来原始文件可以自行进行计算，并同提供的摘要值进行比对，以确保内容没有被修改过。

加解密算法

算法类型	特点	优势	缺陷	代表算法
对称加密	加解密密钥相同或可推算	计算效率高，加密强度高	需提前共享密钥；易泄露	DES、3DES、AES、IDEA
非对称加密	加解密密钥不相关	无需提前共享密钥	计算效率低，仍存在中间人攻击可能	RSA、ElGamal、椭圆曲线系列算法

算法体系

现代加密算法的典型组件包括：加解密算法、加密密钥、解密密钥。其中，加解密算法自身是固定不变的，一般是公开可见的；密钥则往往每次不同，并且需要保护起来，一般来说，对同一种算法，密钥长度越长，则加密强度越大。

加密过程中，通过加密算法和加密密钥，对明文进行加密，获得密文。

解密过程中，通过解密算法和解密密钥，对密文进行解密，获得明文。

根据加解密的密钥是否相同，算法可以分为对称加密（symmetric cryptography，又称公共密钥加密，common-key cryptography）和非对称加密(asymmetric cryptography，又称公钥加密，public-key cryptography)。两种模式适用于不同的需求，恰好形成互补，很多时候也可以组合使用，形成混合加密机制。

并非所有加密算法的强度都可以从数学上进行证明。公认的高强度加密算法是在经过长时间各方面实践论证后，被大家所认可，不代表其不存在漏洞。但任何时候，自行发明加密算法都是一种不太明智的行为。

对称加密

顾名思义，加解密的密钥是相同的。

优点是加解密效率高（速度快，空间占用小），加密强度高。

缺点是参与多方都需要持有密钥，一旦有人泄露则安全性被破坏；另外如何在不安全通道下分发密钥也是个问题。

对称密码从实现原理上可以分为两种：分组密码和序列密码。前者将明文切分为定长数据块作为加密单位，应用最为广泛。后者则只对一个字节进行加密，且密码不断变化，只用在一些特定领域，如数字媒介的加密等。

代表算法包括 DES、3DES、AES、IDEA 等。

- DES（Data Encryption Standard）：经典的分组加密算法，1977 年由美国联邦信息处理标准（FIPS）所采用 FIPS-46-3，将 64 位明文加密为 64 位的密文，其密钥长度为 56 位 + 8 位校验。现在已经很容易被暴力破解。
- 3DES：三重 DES 操作：加密 --> 解密 --> 加密，处理过程和加密强度优于 DES，但现在也被认为不够安全。
- AES（Advanced Encryption Standard）：美国国家标准研究所（NIST）采用取代 DES 成为对称加密实现的标准，1997~2000 年 NIST 从 15 个候选算法中评选 Rijndael 算法（由比利时密码学家 Joan Daemon 和 Vincent Rijmen 发明）作为 AES，标准为 FIPS-197。AES 也是分组算法，分组长度为 128、192、256 位三种。AES 的优势在于处理速度快，整个过程可以数学化描述，目前尚未有有效的破解手段。

适用于大量数据的加解密；不能用于签名场景；需要提前分发密钥。

注：分组加密每次只能处理固定长度的明文，因此过长的内容需要采用一定模式进行加密，《实用密码学》中推荐使用 密文分组链接（*Cipher Block Chain*，*CBC*）、计数器（*Counter*，*CTR*）模式。

非对称加密

非对称加密是现代密码学历史上最为伟大的发明，可以很好的解决对称加密需要的提前分发密钥问题。

顾名思义，加密密钥和解密密钥是不同的，分别称为公钥和私钥。

公钥一般是公开的，人人可获取的，私钥一般是个人自己持有，不能被他人获取。

优点是公私钥分开，不安全通道也可使用。

缺点是加解密速度慢，一般比对称加解密算法慢两到三个数量级；同时加密强度相比对称加密要差。

非对称加密算法的安全性往往需要基于数学问题来保障，目前主要有基于大数质因子分解、离散对数、椭圆曲线等几种思路。

代表算法包括：RSA、ElGamal、椭圆曲线（Elliptic Curve Crytosystems，ECC）系列算法。

- RSA：经典的公钥算法，1978 年由 Ron Rivest、Adi Shamir、Leonard Adleman 共同提出，三人于 2002 年获得图灵奖。算法利用了对大数进行质因子分解困难的特性，但目前还没有数学证明两者难度等价，或许存在未知算法在不进行大数分解的前提下解密。
- Diffie-Hellman 密钥交换：基于离散对数无法快速求解，可以在不安全的通道上，双方协商一个公共密钥。
- ElGamal：由 Taher ElGamal 设计，利用了模运算下求离散对数困难的特性。被应用在 PGP 等安全工具中。
- 椭圆曲线算法（Elliptic curve cryptography，ECC）：现代备受关注的算法系列，基于对

椭圆曲线上特定点进行特殊乘法逆运算难以计算的特性。最早在 1985 年由 Neal Koblitz 和 Victor Miller 分别独立提出。ECC 系列算法一般被认为具备较高的安全性，但加解密计算过程往往比较费时。

一般适用于签名场景或密钥协商，不适于大量数据的加解密。

RSA 算法等已被认为不够安全，一般推荐采用椭圆曲线系列算法。

混合加密机制

即先用计算复杂度高的非对称加密协商一个临时的对称加密密钥（会话密钥，一般相对内容来说要短的多），然后双方再通过对称加密对传递的大量数据进行加解密处理。

典型的场景是现在大家常用的 HTTPS 机制。HTTPS 实际上是利用了 Transport Layer Security/Secure Socket Layer（TLS/SSL）来实现可靠的传输。TLS 为 SSL 的升级版本，目前广泛应用的为 TLS 1.0，对应到 SSL 3.1 版本。

建立安全连接的具体步骤如下：

- 客户端浏览器发送信息到服务器，包括随机数 R1，支持的加密算法类型、协议版本、压缩算法等。注意该过程为明文。
- 服务端返回信息，包括随机数 R2、选定加密算法类型、协议版本，以及服务器证书。注意该过程为明文。
- 浏览器检查带有该网站公钥的证书。该证书需要由第三方 CA 来签发，浏览器和操作系统会预置权威 CA 的根证书。如果证书被篡改作假（中间人攻击），很容易通过 CA 的证书验证出来。
- 如果证书没问题，则用证书中公钥加密随机数 R3，发送给服务器。此时，只有客户端和服务器都拥有 R1、R2 和 R3 信息，基于 R1、R2 和 R3，生成对称的会话密钥（如 AES 算法）。后续通信都通过对称加密进行保护。

数字签名

类似在纸质合同上签名确认合同内容，数字签名用于证实某数字内容的完整性（integrity）和来源（或不可抵赖，non-repudiation）。

一个典型的场景是，A 要发给 B 一个文件（一份信息），B 如何获知所得到的文件即为 A 发出的原始版本？A 先对文件进行摘要，然后用自己的私钥进行加密，将文件和加密串都发给 B。B 收到后文件和加密串，用 A 的公钥来解密加密串，得到原始的数字摘要，跟对文件进行摘要后的结果进行比对。如果一致，说明该文件确实是 A 发过来的，并且文件内容没有被修改过。

HMAC

全称是 Hash-based Message Authentication Code，即“基于 Hash 的消息认证码”。基本过程为对某个消息，利用提前共享的对称密钥和 Hash 算法进行加密处理，得到 HMAC 值。该 HMAC 值提供方可以证明自己拥有共享的对称密钥，并且消息自身可以利用 HMAC 确保未经篡改。

HMAC(K, H, Message)

其中，K 为提前共享的对称密钥，H 为提前商定的 Hash 算法（一般为公认的经典算法），Message 为要处理的消息内容。如果不知道 K 和 H，则无法根据 Message 得到准确的 HMAC 值。

HMAC 一般用于证明身份的场景，如 A、B 提前共享密钥，A 发送随机串给 B，B 对称加密处理后把 HMAC 值发给 A，A 收到了自己再重新算一遍，只要相同说明对方确实是 B。

HMAC 主要问题是需要共享密钥。当密钥可能被多方拥有的场景下，无法证明消息确实来自某人（Non-repudiation）。反之，如果采用非对称加密方式，则可以证明。

盲签名

1983 年由 David Chaum 提出。签名者在无法看到原始内容的前提下对信息进行签名。

盲签名主要是为了实现防止追踪（unlinkability），签名者无法将签名内容和结果进行对应。典型的实现包括 RSA 盲签名)。

多重签名

n 个持有人中，收集到至少 m 个（ ）的签名，即认为合法，这种签名被称为多重签名。

其中，n 是提供的公钥个数，m 是需要匹配公钥的最少的签名个数。

群签名

环签名

环签名由 Rivest,shamir 和 Tauman 三位密码学家在 2001 年首次提出。环签名属于一种简化的群签名。

签名者首先选定一个临时的签名者集合,集合中包括签名者自身。然后签名者利用自己的私钥和签名集合中其他人的公钥就可以独立的产生签名,而无需他人的帮助。签名者集合中的其他成员可能并不知道自己被包含在其中。

数字证书

数字证书用来证明某个公钥是谁的，并且内容是正确的。

对于非对称加密算法和数字签名来说，很重要的一点就是公钥的分发。一旦公钥被人替换（典型的如中间人攻击），则整个安全体系将被破坏掉。

怎么确保一个公钥确实是某个人的原始公钥？

这就需要数字证书机制。

顾名思义，数字证书就是像一个证书一样，证明信息和合法性。由证书认证机构（Certification Authority，CA）来签发，权威的 CA 包括 verisign 等。

数字证书内容可能包括版本、序列号、签名算法类型、签发者信息、有效期、被签发人、签发的公开密钥、**CA** 数字签名、其它信息等等，一般使用最广泛的标准为 ITU 和 ISO 联合制定的 X.509 规范。

其中，最重要的包括 `签发的公开密钥`、`CA 数字签名` 两个信息。因此，只要通过这个证书就能证明某个公钥是合法的，因为带有 CA 的数字签名。

更进一步地，怎么证明 CA 的签名合法不合法呢？

类似的，CA 的数字签名合法不合法也是通过 CA 的证书来证明的。主流操作系统和浏览器里面会提前预置一些 CA 的证书（承认这些是合法的证书），然后所有基于他们认证的签名都会自然被认为合法。

后面章节将介绍的 PKI 体系提供了一套完整的证书管理的框架。

PKI 体系

在非对称加密中，公钥则可以通过证书机制来进行保护，如何管理和分发证书则可以通过 PKI（Public Key Infrastructure）来保障。

顾名思义，PKI 体系在现代密码学应用领域处于十分基础的地位，解决了十分核心的证书管理问题。

PKI 并不代表某个特定的密码学技术和流程，PKI 是建立在公私钥基础上实现安全可靠传递消息和身份确认的一个通用框架。实现了 PKI 的平台可以安全可靠地管理网络中用户的密钥和证书，包括多个实现和变种，知名的有 RSA 公司的 PKCS（Public Key Cryptography Standards）标准和 X.509 规范等。

一般情况下，PKI 至少包括如下组件：

- CA（Certification Authority）：负责证书的颁发和作废，接收来自 RA 的请求，是最核心的部分；
- RA（Registration Authority）：对用户身份进行验证，校验数据合法性，负责登记，审核过了就发给 CA；
- 证书数据库：存放证书，一般采用 LDAP 目录服务，标准格式采用 X.500 系列。

CA 是最核心的组件，主要完成对证书的管理。

常见的流程为，用户通过 RA 登记申请证书，CA 完成证书的制造，颁发给用户。用户需要撤销证书则向 CA 发出申请。

之前章节内容介绍过，密钥有两种类型：用于签名和用于加解密，对应称为 `签名密钥对` 和 `加密密钥对` 。

用户证书可以有两种方式。一般可以由 CA 来生成证书和私钥；也可以自己生成公钥和私钥，然后由 CA 来对公钥进行签发。后者情况下，当用户私钥丢失后，CA 无法完成恢复。

Merkle 树

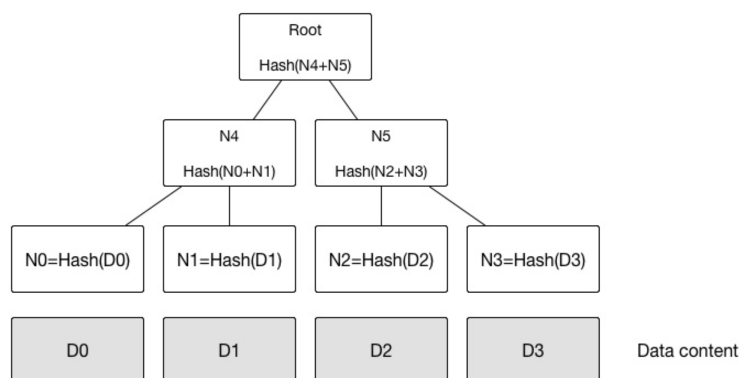


图 *1.8.6.1 - Merkle* 树示例

默克尔树（又叫哈希树）是一种二叉树，由一个根节点、一组中间节点和一组叶节点组成。最下面的叶节点包含存储数据或其哈希值，每个中间节点是它的两个孩子节点内容的哈希值，根节点也是由它的两个子节点内容的哈希值组成。

进一步的，默克尔树可以推广到多叉树的情形。

默克尔树的特点是，底层数据的任何变动，都会传递到其父亲节点，一直到树根。

默克尔树的典型应用场景包括：

- 快速比较大量数据：当两个默克尔树根相同时，则意味着所代表的数据必然相同。
- 快速定位修改：例如上例中，如果 D1 中数据被修改，会影响到 N1，N4 和 Root。因此，沿着 Root --> N4 --> N1，可以快速定位到发生改变的 D1；
- 零知识证明：例如如何证明某个数据（D0……D3）中包括给定内容 D0，很简单，构造一个默克尔树，公布 N0，N1，N4，Root，D0 拥有者可以很容易检测 D0 存在，但不知道其它内容。

同态加密

定义

同态加密（Homomorphic Encryption）是一种特殊的加密方法，允许对密文进行处理得到仍然是加密的结果，即对密文直接进行处理，跟对明文进行处理再加密，得到的结果相同。从代数的角度讲，即同态性。

如果定义一个运算符 ，对加密算法 `E` 和 解密算法 `D`，满足：

则意味着对于该运算满足同态性。

同态性在代数上包括：加法同态、乘法同态、减法同态和除法同态。同时满足加法同态和乘法同态，则意味着是 `代数同态`，即 `全同态`。同时满足四种同态性，则被称为 `算数同态`。

历史

同态加密的问题最早是由 Ron Rivest、Leonard Adleman 和 Michael L. Dertouzos 在 1978 年提出，但 第一个“全同态”的算法 到 2009 年才被克雷格·金特里（Craig Gentry）证明。

仅满足加法同态的算法包括 Paillier 和 Benaloh 算法；仅满足乘法同态的算法包括 RSA 和 ElGamal 算法。

同态加密在云时代的意义十分重大。目前，从安全角度讲，用户还不敢将敏感信息直接放到第三方云上进行处理。如果有了比较实用的同态加密技术，则大家就可以放心的使用各种云服务了。

遗憾的是，目前已知的同态加密技术需要消耗大量的计算时间，还远达不到实用的水平。

函数加密

与同态加密相关的一个问题是函数加密。

同态加密保护的是数据本身，而函数加密顾名思义保护的是处理函数本身，即让第三方看不到处理过程的前提下，对数据进行处理。

该问题已被证明是不存在对多个通用函数的任意多 key 的方案，目前仅能做到对某个特定函数的一个 key 的方案。

其它问题

零知识证明（zero knowledge validation）

证明者在不向验证者提供任何有用的信息的前提下，使验证者相信某个论断是正确的。

例如，A 向 B 证明自己有一个物品，但 B 无法拿到这个物品，无法用 A 的证明去向别人证明自己也拥有这个物品。

小结

比特币项目

做设计，很多时候都是在权衡 **trade-off**。

比特币项目是区块链技术首个大规模的成功应用，并且是首个得到实践检验的数字货币实现，在金融学和信息技术历史上都具有十分重要的意义。

本章将介绍其来源、原理设计和相关的工具和技术点等。

简介

比特币是基于密码学和经济博弈的一种数字货币，也是历史上首个经过大规模长时间运作检验的数字货币系统。

从 blockchain.info 网站 可以从查询到比特币的汇率（以美元为单位）变化历史。



图 *1.9.1.1* - 比特币汇率历史

历史

2008 年 10 月 31 日，中本聪发布比特币唯一的白皮书：《Bitcoin：A Peer-to-Peer Electronic Cash System/比特币：一种点对点的电子现金系统》。

2009 年 1 月 3 日，中本聪在位于芬兰赫尔辛基的一个小型服务器上挖出了第一批 50 个比特币，并记录下当天泰晤士报的头版标题：“ The Times 03/Jan/2009 Chancellor on brink of second bailout for banks ”。

2010 年 5 月 21 日，第一次比特币交易：佛罗里达程序员 Laszlo Hanyecz 用 1 万 BTC 购买了价值 25 美元的披萨优惠券。这是比特币的首个兑换汇率：1: 0.0025 美金。这些比特币在今日价值约 700 万美金。

2010 年 7 月 17 日，第一个比特币平台成立。

2011 年，开始出现基于显卡的挖矿设备。2011 年底，汇率约为 2 美元。

2012 年 9 月 27 日，比特币基金创立，此时比特币价格为 12.46 美元。

2012 年 11 月 28 日，比特币产量第一次减半。

2013 年 3 月，1/3 的专业矿工已经采用专用 ASIC 矿机进行挖矿。

2013年 4 月 10 日，BTC 创下历史最高价，266 美元。

2013 年 6 月 27日，德国议会作出决定：持有比特币一年以上将予以免税，被业内认为此举变相认可了比特币的法律地位，此时比特币价格为 102.24 美元。

2013 年 10 月，世界第一台可以兑换比特币的 ATM 在加拿大上线。

2013 年 11 月 29 日，比特币的交易价格创下 1242 美元的历史新高，而同时黄金价格为一盎司 1241.98 美元，比特币价格首度超过黄金。

2014 年 2 月，全球最大比特币交易平台 Mt.Gox 宣告因 85 万个比特币被盗而破产并关闭，造成大量投资者的损失，比特币价格一度暴跌。

2014 年 3 月，中国第一台可以兑换比特币的 ATM 在香港上线。

2014 年 6 月，美国加州通过 AB-129 法案，允许比特币等数字货币在加州进行流通。

2015 年 6 月，纽约成为美国第一个正式进行数字货币监管的州。

2015 年 10 月，欧盟法院裁定比特币交易免征增值税。

2016 年 1 月，中国人民银行在京召开了数字货币研讨会，会后发布公告宣称或推出数字货币。

2016 年 7 月 9 日，比特币产量第二次减半。

时至今日，比特币汇率约为 600 美元，总市值在 100 亿 美金。八成的交易量在中国。

比特币区块链目前生成了约 42 万个区块，完整存储需要约 75 GB 的空间。主流的交易所包括 Bitstamp、BTC-e、Bitfinex 等。多家投资机构（包括红杉、IDG、软银、红点等）都有布局。

注：通过 *blockchain.info* 可以实时查询到更多详细数据。

山寨币

比特币的“成功”，刺激了相关的生态和社区发展，大量类似数字货币（超过 700 种）纷纷出现，被称为“山寨币”，比较出名的包括以太币和瑞波（Ripple）币。



这些山寨币，要么建立在独立的区块链上，要么复用已有的区块链（例如比特币）。

原理和设计

比特币网络是一个分布式的点对点网络，网络中的矿工通过“挖矿”来完成对交易记录的记账过程，维护网络的正常运行。

比特币通过区块链网络提供一个公共可见的记账本，用来记录发生过的交易的历史信息。

每次发生交易，用户需要将新交易记录写到比特币区块链网络中，等网络确认后即可认为交易完成。每个交易包括一些输入和一些输出，未经使用的交易的输出（ Unspent Transaction Outputs，UTXO）可以被新的交易引用作为合法的输入。

一笔合法的交易，即引用某些已存在交易的 UTXO，作为交易的输入，并生成新的输出的过程。

在交易过程中，转账方需要通过签名脚本来证明自己是 UTXO 的合法使用者，并且指定输出脚本来限制未来对本交易的使用者（为收款方）。对每笔交易，转账方需要进行签名确认。并且，对每一笔交易来说，总输入不能小于总输出。

交易的最小单位是“聪”，即 比特币。

下图展示了一些简单的示例交易。更一般情况下，交易的输入输出可以为多方。

交易	目的	输入	输出	签名	差额
T0	A 转给 B	别人给 A 的交易的输出	B 账户可以使用该交易	A 签名确认	输入减输出，为交易服务费
T1	B 转给 C	T0 的输出	C 账户可以使用该交易	B 签名确认	输入减输出，为交易服务费
...	X 转给 Y	别人给 X 的交易的输出	Y 账户可以使用该交易	X 签名确认	输入减输出，为交易服务费

下面分别介绍比特币网络中的重要概念和设计思路。

概念

账户/地址

比特币账户采用了非对称的加密算法，用户自己保留私钥，对他发出的交易进行签名确认，并公开公钥。

比特币的账户地址其实就是用户公钥经过一系列 hash（HASH160，或先进行 SHA256，然后进行 RIPEMD160）及编码运算后生成的 160 位（20 字节）的字符串。

一般，也常常对账户地址串进行 Base58Check 编码，并添加前导字节（表明支持哪种脚本）和 4 字节校验字节，以提高可读性和准确性。

注：这里账户并非直接是公钥，而是 *hash* 后的值，避免公钥过早暴露导致被破解出私钥。

交易

交易是完成比特币功能的核心概念，一条交易将可能包括如下信息：

- 付款人地址：合法的地址，公钥经过 SHA256 和 RIPEMD160 两次 hash，得到 160 位 hash 串；
- 付款人对交易的签字确认：确保交易内容不被篡改；
- 付款人资金的来源交易 ID：从哪个交易的输出作为本次交易的输入；
- 交易的金额：多少钱，跟输入的差额为交易的服务费；
- 收款人地址：合法的地址；
- 收款人的公钥：收款人的公钥；
- 时间戳：交易何时能生效。

网络中节点收到交易信息后，将进行如下检查：

- 交易是否已经处理过；
- 交易是否合法。包括地址是否合法、发起交易者是输入地址的合法拥有者、是否是 UTXO；
- 交易的输入之和是否大于输出之和。

检查都通过，则将交易标记为合法的未确认交易，并在网络内进行广播。

可以从 blockchain.info 网站查看实时的交易信息。例如一次较新的交易 0beca08914de596217f098d744e3fb8da68aa5e00dd8f63a3364b451f3f4a70f。

脚本

脚本（Script） 是保障交易完成（主要用于检验交易是否合法）的核心机制，当所依附的交易发生时被触发。通过脚本机制而非写死交易过程，比特币网络实现了一定的可扩展性。比特币脚本语言是一种非图灵完备的语言，类似 Forth 语言。

一般每个交易都会包括两个脚本：输出脚本（scriptPubKey）和认领脚本（scriptSig）。

输出脚本一般由付款方对交易设置锁定，用来对能动用这笔交易输出（例如，要花费交易的输出）的对象（收款方）进行权限控制，例如限制必须是某个公钥的拥有者才能花费这笔交易。

认领脚本则用来证明自己可以满足交易输出脚本的锁定条件，即对某个交易的输出（比特币）的拥有权。

输出脚本目前支持两种类型：

- P2PKH：Pay-To-Public-Key-Hash，允许用户将比特币发送到一个或多个典型的比特币地址上（证明拥有该公钥），前导字节一般为 0x00；
- P2SH：Pay-To-Script-Hash，支付者创建一个输出脚本，里边包含另一个脚本（认领脚本）的哈希，一般用于需要多人签名的场景，前导字节一般为 0x05；

以 P2PKH 为例，输出脚本的格式为

```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

其中，OP_DUP 是复制栈顶元素；OP_HASH160 是计算 hash 值；OP_EQUALVERIFY 判断栈顶两元素是否相等；OP_CHECKSIG 判断签名是否合法。这条指令实际上保证了只有 pubKey 的拥有者才能合法引用这个输出。

另外一个交易如果要花费这个输出，在引用这个输出的时候，需要提供认领脚本格式为

```
scriptSig: <sig> <pubKey>
```

其中， 是拿 pubKey 对应的私钥对交易（全部交易的输出、输入和脚本）hash 值进行签名，pubKey 的 hash 值需要等于 pubKeyHash。

进行交易验证时，会按照先 scriptSig 后 scriptPubKey 的顺序进行依次入栈处理，即完整指令为：

```
<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

读者可以按照栈的过程来进行计算，体会脚本的验证过程。

引入脚本机制带来了灵活性，但也引入了更多的安全风险。比特币脚本支持的指令集十分简单，基于栈的处理方式，并且非图灵完备，此外还添加了额外的一些限制（大小限制等）。

区块

一个区块将主要包括如下内容：

4 字节的区块大小信息；

80 字节的区块头信息：

- 版本号：4 字节；
- 上一个区块头的 SHA256 hash 值：链接到一个合法的块上，32 字节；
- 包含的所有验证过的交易的 Merkle 树根的哈希值，32 字节；
- 时间戳：4 字节；

- 难度指标：4 字节；
- Nonce：4 字节，PoW 问题的答案；

交易个数计数器：1~9 字节；

所有交易的具体内容，可变长。

设计理念

如何避免作恶

基于经济博弈原理。在一个开放的网络中，无法通过技术手段保证每个人都是合作的。但可以通过经济博弈来让合作者得到利益，让非合作者遭受损失和风险。

实际上，博弈论早已被广泛应用到众多领域。

一个经典的例子是两个人来分一个蛋糕，如果都想拿到较大的一块，在没有第三方的前提下，该怎么指定规则才公平？

最简单的一个方案是负责切蛋糕的人后选。

注：如果推广到 *N* 个人呢？

比特币网络需要所有试图参与者（矿工）都首先要付出挖矿的代价，进行算力消耗，越想拿到新区块的决定权，意味着抵押的算力越多。一旦失败，这些算力都会被没收掉，成为沉没成本。当网络中存在众多参与者时，个体试图拿到新区块决定权要付出的算力成本是巨大的，意味着进行一次作恶付出的代价已经超过可能带来的好处。

负反馈调节

比特币网络在设计上，很好的体现了负反馈的控制论基本原理。

比特币网络中矿工越多，系统就越稳定，比特币价值就越高，但挖到矿的概率会降低。

反之，网络中矿工减少，会让系统更容易导致被攻击，比特币价值越低，但挖到矿的概率会提高。

因此，比特币的价格理论上应该稳定在一个合适的值（网络稳定性也会稳定在相应的值），这个价格乘以挖到矿的概率，恰好达到矿工的收益预期。

从长远角度看，硬件成本是下降的，但每个区块的比特币奖励每隔 4 年减半，最终将在 2140 年达到 2100 万枚，之后将完全依靠交易的服务费来鼓励矿工对网络的维护。

注：比特币最小单位是“聪”，即 10^{-8}，总“聪”数为 $2.1E15$。对于 *64* 位处理器来说，高精度浮点计数的限制导致单个数值不能超过 。

共识机制

传统的共识问题是考虑在一个相对封闭的体系中，存在好节点、坏节点，然后如何达成一致。

对于比特币网络来说，因为它是开放的，网络质量也是完全无法保证的，导致问题更加复杂，难以依靠传统的一致性算法来实现。

比特币网络对共识进行了一系列的放宽，同时对参与共识进行了一系列的限制。

首先是不实现最终共识，理论上现有达成的任何结果都可能被推翻，只是被推翻的可能性随着时间而指数级的下降，要付出的代价迅速上升。

此外，达成共识的时间比较长，而且是按照块来进行阶段性的确认（快照），提高网络可用性。

此外，通过进行 PoW 限制合法提案的个数，提高网络的稳定性。

挖矿

原理与过程

了解比特币，最应该知道的一个概念就是“挖矿”，挖矿是参与维护比特币网络的节点，通过协助生成新区块来获取一定量新增的比特币。

当用户发布交易后，需要有人将交易进行确认，写到区块链中，形成新的区块。在一个互相不信任的系统中，该由谁来完成这件事情呢？比特币网络采用了“挖矿”的方式来解决这个问题。

目前，每 10 分钟左右生成一个不超过 1 MB 大小的区块（记录了这 10 分钟内发生的验证过的交易内容），串联到最长的链尾部，每个区块的成功提交者可以得到系统 12.5 个比特币的奖励（一定区块数后才能使用），以及用户附加到交易上的支付服务费用。

注：每个区块的奖励一开始是 *50* 个比特币，每隔 *21* 万个区块自动减半，即 *4* 年时间，最终比特币总量稳定在 *2100* 万个。因此，比特币是一种通缩的货币。

挖矿的具体过程为：参与者根据上一个区块的 hash 值，10 分钟内的验证过的交易内容，再加上自己猜测的一个随机数 X，让新区块的 hash 值小于比特币网络中给定的一个数。这个数越小，计算出来就越难。系统每隔两周（即经过 2016 个区块）会根据上一周期的挖矿时间来调整挖矿难度（通过调整限制数的大小），来调节生成区块的时间稳定在 10 分钟左右。为了避免震荡，每次调整的最大幅度为 4 倍。

为了挖到矿，参与处理区块的用户端往往需要付出大量的时间和计算力。算力一般以每秒进行多少次 hash 计算为单位，记为 h/s。

> 汇丰银行分析师 Anton Tonev 和 Davy Jose 表示，比特币区块链（通过挖矿）提供了一个局部的、迄今为止最优的解决方案：如何在分散的系统中验证信任。这就意味着，区块链本质上解决了传统依赖于第三方的问题，因为这个协议不止满足了中心化机构追踪交易的需求，还使得陌生人之间产生信任。区块链的技术和安全的过程使得陌生人之间在没有被信任的第三方时产生信任。

如何看待挖矿

2010 年左右，挖矿还是一个很有前途的行业。但是现在，建议还是不要考虑了，因为从概率上说，由于当前参与挖矿的计算力实在过于庞大（已经超出了大部分的超算中心），获得比特币的收益已经眼看要 cover 不住电费了。特别那些想着用云计算虚拟机来挖矿的想法，意义确实不大了。

从普通的 CPU（2009 年）、到后来的 GPU（2010 年）和 FPGA（2011 年末）、到后来的 ASIC 矿机（2013 年初，目前单片算力已达每秒数百亿次 Hash 计算）、再到现在众多矿机联合组成矿池。短短数年间，比特币矿机的技术走完了过去几十年的集成电路技术进化历程，并且还颇有创新之处。确实是哪里有利益，哪里的技术就飞速发展！目前，矿机主要集中在中国大陆（超过一半的算力）和欧美，大家比拼的是一定计算性能情况下低电压和低功耗的电路设计。全网的算力已超过每秒 10^{18} 次 Hash 计算。

很自然的，有人会想到，如果我有很强大的计算力，所有的块都是我算出来了，拒不承认别人的交易内容，那是不是就能破坏比特币网络。确实如此，基本上拿到 1/3 的计算力，比特币网络就存在被破坏的风险了；拿到 1/2，概率上就掌控整个网络了。但是这个将需要付出巨大的计算成本。

那么有没有办法防护呢？除了尽量避免计算力放到同一个组织手里，没太好的办法，这是目前 PoW（Proof of Work）的协议规定的。

也有人觉得为了算出一个块，大部分计算力（特别是没算出来的算力）其实都浪费了。有人提出用所谓的 PoS（Proof of Stake）和 DPoS，即大节点作为多个节点代理人的模式来节约计算力。那怎么选大节点？又容易导致“富则越富”问题。这其实就是完全民主 vs 选举人制度嘛。

个人认为，无论 PoW 还是 PoS，都无法解决所有问题。要从根本上解决，得引入随机代理人制度，通过算法在某段时间内只让部分节点参加计算，并且要发放一部分“普世奖励”给所有在线节点。

工具

客户端

客户端分为三种：完整客户端、轻量级客户端和在线客户端。

- 完整客户端：存储所有的交易历史记录，功能完备；
- 轻量级客户端：不保存交易副本，交易需要向别人查询；
- 在线客户端：通过网页模式来浏览第三方服务器提供的服务。

钱包

矿机

专门为"挖矿"设计的硬件，包括基于 GPU 和 ASIC 的芯片。

脚本

比特币交易支持一种比较简单的脚本语言（类 Forth 的栈脚本语言），可以写入 UTXO。交易发生时，输入的解锁脚本和输出的锁定脚本进行执行，检验交易合法性。

比特币脚本并不支持循环等复杂的流控制，因此它是非图灵完备的。

共识机制

比特币网络是公开的，因此共识协议的稳定性和防攻击性十分关键。

比特币区块链采用了 Proof of Work（PoW）的机制来实现共识，该机制于 1998 年在 B-money 设计中提出。

目前，Proof of 系列中比较出名的一致性协议包括 PoW 和 PoS，都是通过经济惩罚来限制恶意参与。

PoW

工作量证明，Proof of Work，通过计算来猜测一个数值（nonce），得以解决规定的 hash 问题（来源于 hashcash）。保证在一段时间内，系统中只能出现少数合法提案。

同时，这些少量的合法提案会在网络中进行广播，收到的用户进行验证后会基于它认为的最长链上继续难题的计算。因此，系统中可能出现链的分叉（Fork），但最终会有一条链成为最长的链。

hash 问题具有不可逆的特点，因此，目前除了暴力计算外，还没有有效的算法进行解决。反之，如果获得符合要求的 nonce，则说明在概率上是付出了对应的算力。谁的算力多，谁最先解决问题的概率就越大。当掌握超过全网一半算力时，从概率上就能控制网络中链的走向。这也是所谓 `51% 攻击` 的由来。

参与 PoW 计算比赛的人，将付出不小的经济成本（硬件、电力、维护等）。当没有成为首个算出的“幸运儿”时，这些成本都将被沉没掉。这也保障了，如果有人恶意破坏，需要付出大量的经济成本。也有设计试图将后算出结果者的算力按照一定比例折合进下一轮比赛考虑。

有一个很直观的例子可以说明为何这种经济博弈模式会确保系统中最长链的唯一。

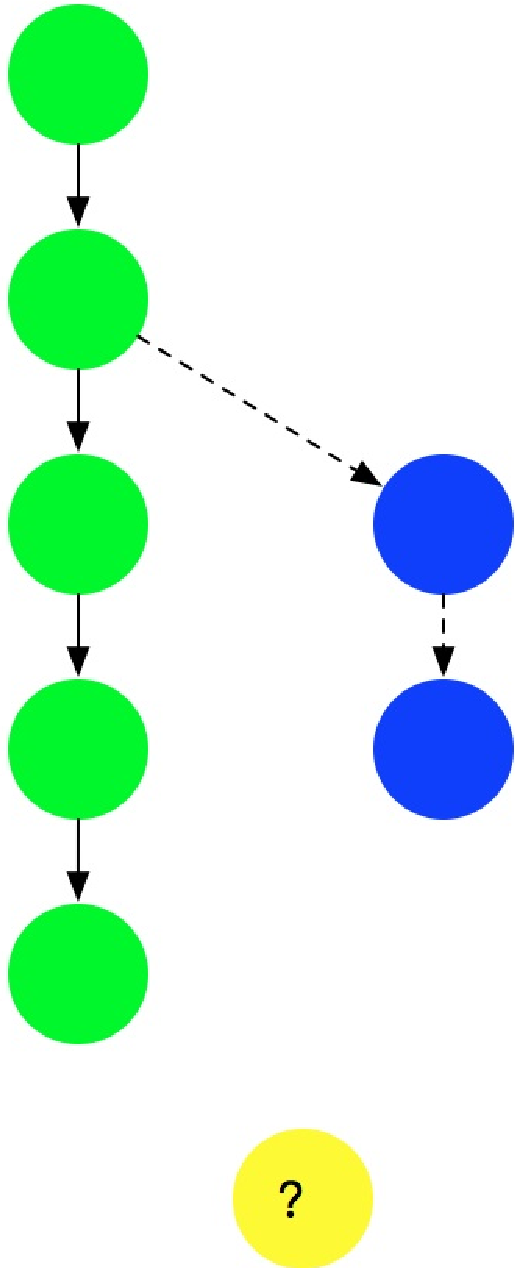


图 *1.9.5.1 - PoW* 保证一致性

超市付款需要排成一队，可能有人不守规矩要插队。超市管理员会检查队伍，认为最长的一条队伍是合法的，并让不合法的分叉队伍重新排队。只要大部分人不傻，就会自觉在最长的队伍上排队。

PoS

权益证明，Proof of Stake，2013 年被提出，最早在 Peercoin 系统中被实现，类似现实生活中的股东机制，拥有股份越多的人越容易获取记账权。

典型的过程是通过保证金（代币、资产、名声等具备价值属性的物品即可）来对赌一个合法的块成为新的区块，收益为抵押资本的利息和交易服务费。提供证明的保证金（例如通过转账货币记录）越多，则获得记账权的概率就越大。合法记账者可以获得收益。

PoS 是试图解决在 PoW 中大量资源被浪费的缺点。恶意参与者将存在保证金被罚没的风险，即损失经济利益。

一般的，对于 PoS 来说，需要掌握超过全网 □ 的资源，才有可能左右最终的结果。这个也很容易理解，三个人投票，前两人分别支持一方，这时候，第三方的投票将决定最终结果。

PoS 也有一些改进的算法，包括授权股权证明机制（DPOS），即股东们投票选出一个董事会，董事会中成员才有权进行代理记账。

闪电网络

比特币的交易网络最为人诟病的一点便是交易性能：全网每秒 7 笔的交易速度，远低于传统的金融交易系统；同时，等待 6 个块的可信确认导致约 1 个小时的最终确认时间。

闪电网络的主要思路十分简单 -- 将大量交易放到比特币区块链之外进行。该设计最早是 2015 年 2 月在论文《The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments》中提出。

比特币的区块链机制自身提供了很好的可信保障，但是很慢；另一方面考虑，对于大量的小额交易来说，是否真实需要这么高的可信性？闪电网络通过智能合约来完善链下的交易渠道。

核心的概念主要有两个：RSMC（Recoverable Sequence Maturity Contract）和 HTLC（Hashed Timelock Contract）。前者解决了链下交易的确认问题，后者解决了支付通道的问题。

RSMC

Recoverable Sequence Maturity Contract，中文可以翻译为“可撤销的顺序成熟度合同”。这个词很绕，其实主要原理很简单，就是类似准备金机制。

我们先假定交易双方之间存在一个“微支付通道”（资金池）。双方都预存一部分资金到“微支付通道”里，之后每次交易，就对交易后的资金分配方案共同进行确认，同时签字作废旧的版本。当需要提现时，将最终交易结果写到区块链网络中，被最终确认。可以看到，只有在提现时候才需要通过区块链。

任何一个版本的方案都需要经过双方的签名认证才合法。任何一方在任何时候都可以提出提现，提现需要提供一个双方都签名过的资金分配方案（意味着肯定是某次交易后的结果）。在一定时间内，如果另外一方提出证明表明这个方案其实之前被作废了（非最新的交易结果），则资金罚没给质疑成功方。这就确保了没人会拿一个旧的交易结果来提现。

另外，即使双方都确认了某次提现，首先提出提现一方的资金到账时间要晚于对方，这就鼓励大家尽量都在链外完成交易。

HTLC

微支付通道是通过 Hashed Timelock Contract 来实现的，中文意思是“哈希的带时钟的合约”。这个其实就是限时转账。理解起来其实也很简单，通过智能合约，双方约定转账方先冻结一笔钱，并提供一个哈希值，如果在一定时间内有人能提出一个字符串，使得它哈希后的值跟已知值匹配（实际上意味着转账方授权了接收方来提现），则这笔钱转给接收方。

不太恰当的例子，约定一定时间内，有人知道了某个暗语（可以生成匹配的哈希值），就可以拿到这个指定的资金。

推广一步，甲想转账给丙，丙先发给甲一个哈希值。甲可以先跟乙签订一个合同，如果你在一定时间内能告诉我一个暗语，我就给你多少钱。乙于是跑去跟丙签订一个合同，如果你告诉我那个暗语，我就给你多少钱。丙于是告诉乙暗语，拿到乙的钱，乙又从甲拿到钱。最终达到结果是甲转账给丙。这样甲和丙之间似乎构成了一条完整的虚拟的“支付通道”。

HTLC 的机制可以扩展到多个人，大家可以想象一下，想象出来了就理解了闪电网络。

闪电网络

RSMC 保障了两个人之间的直接交易可以在链下完成，HTLC 保障了任意两个人之间的转账都可以通过一条“支付”通道来完成。整合这两种机制，就可以实现任意两个人之间的交易都可以在链下完成了。

在整个交易中，智能合约起到了中介的重要角色，而区块链则确保最终的交易结果被确认。

侧链

允许资产在比特币区块链和其它链之间互转。降低核心的区块链上发生交易的次数。

也来自比特币社区，2013 年 12 月提出，2014 年 4 月成立项目。

通过简单地复用现有比特币的方式，实现比特币和其他帐簿资产在多个区块链间的转移。

Blockstream 基于侧链技术探索更多功能，已发布商业化应用 Liquid，并与普华永道进行相关合作。

小结

本章介绍了比特币的相关知识。比特币作为数字货币领域的重大突破，对分布式记账领域有着很深远的影响。

虽然在隐私保护等方面，比特币仍然为人诟病，但其底层的区块链技术已经受到重视，在许多方面都具有技术优势。

细分来看，比特币网络系统中并没有特殊创新的技术，它有机的组合了如下领域的已有成果：

- 密码学
- 博弈论
- 记账技术
- 分布式系统
- 控制论

甚至可以说，对这些技术的应用并没有达到十分专业的地步。

但正是如此巧妙地组合，让它能完成这样一件了不起的创举。

这或许就是“大师”与“专家”境界的些许差异。

Ethereum - 以太坊项目

君子和而不同。

以太坊项目进一步扩展了区块链网络的能力，从交易延伸为智能合约（Smart Contract）。

其官网首页为 ethereum.org。

简介

根据以太坊官方的宣称，以太坊（Ethereum）目标是打造成一个运行智能合约的去中心化平台（Platform for Smart Contract），平台上的应用按程序设定运行，不存在停机、审查、欺诈、第三方人为干预的可能。以太坊平台由 Golang、C++、Python 等多种编程语言实现。

当然，为了打造这个平台，以太坊提供了一条公开的区块链，并制定了面向智能合约的一套编程语言。智能合约开发者可以在其上使用官方提供的工具来开发支持以太坊区块链协议的应用（即所谓的 DAPP）。

历史与规划

2014 年，以太坊项目开始众筹计划。

2015 年 7 月，众筹完成，筹到价值 1800 万美金的比特币，第一阶段 Frontier 发布，以太坊区块链网络正式上线。

2016 年 3 月，第二阶段 Homestead 开始运行（区块数 1150000），主要改善了安全性。

2017 年 3Q，发布 Metropolis；

2018 年 1Q，发布 Serenity，发布区块链的 PoS 股权证明(Casper)版本。

特点

以太坊区块链的特点主要包括：

- 单独为智能合约指定编程语言 Solidity；
- 使用了内存需求较高的哈希函数：避免出现算力矿机；
- uncle 块激励机制：降低矿池的优势，减少区块产生间隔为 15 秒；
- 难度调整算法：一定的自动反馈机制；
- gas 限制调整算法：限制代码执行指令数，避免循环攻击；
- 记录当前状态的哈希树的根哈希值到区块：某些情形下实现轻量级客户端；
- 为执行智能合约而设计的简化的虚拟机 EVM。

组织

- 以太坊基金会：2014 年 6 月在瑞士注册的非营利性机构，管理以太坊获得的资金分配。

安装部署

如果你是首次接触 ethereum，推荐使用下面的步骤安装部署。

安装 Go 环境

```
curl -O https://storage.googleapis.com/golang/go1.5.1.linux-amd64.tar.gz
tar -C /usr/local -xzf go1.5.1.linux-amd64.tar.gz
mkdir -p ~/go; echo "export GOPATH=$HOME/go" >> ~/.bashrc
echo "export PATH=$PATH:$HOME/go/bin:/usr/local/go/bin" >> ~/.bashrc
source ~/.bashrc
```

安装 ethereum

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo add-apt-repository -y ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install ethereum
```

安装 solc 编译器

```
sudo add-apt-repository ppa:ethereum/ethereum-qt
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install cpp-ethereum
```

安装后可以使用 geth 命令创建 ethereum 账户

```
geth account new
```

Solidity 语言支持

browser-solidity 提供了在线的 Solidity 语言测试。

用户也可以从 这里 下载到包括 Solidity 运行环境的安装包。

相关工具

客户端

官方提供钱包客户端 Mist，支持进行交易，同时支持直接编写和部署智能合约。

所编写的代码编译发布后，可以部署到区块链上。使用者可通过发送调用相应合约方法的交易，由矿工的以太坊虚拟机（EVM）在区块链上执行。

以太坊现在有多种语言实现的客户端，包括：

- ethereumjs-lib：javascript 语言实现；
- Ethereum(J)：Java 语言实现；
- ethereumH：Haskell 语言实现；
- go-ethereum：go 语言实现；
- Parity：Rust 语言实现；
- pyethapp：python 语言实现；
- ruby-ethereum：Ruby 语言实现；

IDE

网站资源

已有一些网站提供对以太坊网络的数据查看，包括 EthStats.net、EtherNodes.com 等。

协议设计

核心概念

- EVM：以太坊虚拟机，轻量级虚拟机环境，是以太坊中智能合约的运行环境。
- Account：账户，分两类：合约账户存储执行的合约代码；外部账户为以太币拥有者账户，对应到某公钥。
- Transaction：交易，从一个账户到另一个账户的消息，包括以太币或者合约执行参数。
- Gas：燃料，每执行一条合约指令会消耗一定的燃料，当某个交易还未执行结束，而燃料消耗完时，合约执行终止并回滚状态。

一致性

目前采用了 PoW 作为一致达成保证，未来可能迁移到 PoS 上。

降低攻击

设计核心思想是通过经济激励机制防止少数人作恶：

- 所有交易都要提供交易费用，避免 DDoS 攻击；
- 程序运行指令数通过 gas 来限制，所消耗的费用超过设定上限时会被取消，避免恶意合约。

提高扩展性

以太坊未来希望通过分片机制可以提高整个网络的扩展性。分片之前整个网络的处理取决于单个节点的处理。

分片后，只有同一片内的处理是同步的、一致的，不同分片之间则可以是异步的。

链码示例一：Hello World!

简介

smartContract_example01.sol

合约greeter是一个简单的智能合约，你可以使用这个合约来和其他人交流，它的回复会同你的输入完全一样，当输入为“Hello World!”的时候，合约也会回复“Hello World!”。

目的:

该合约主要面向第一次接触solidity和ethereum的初学者,旨在让大家能够了解如何编写一个简单的智能合约程序,掌握基本流程。

主要实现如下的功能：

- 返回你预先设置的字符串

主要函数

- `kill` ：selfdestruct 是 ethereum 智能合约自带的自毁程序,kill对此方法进行了封装,只有合约的拥有者才可以调用该方法；
- `greet` ：返回合约 greeter 里的 greeting属性的值；

代码运行分析

第一步 生成智能合约代码对象

我们先把合约代码smartContract_example01.sol 压缩为一行．新建一个ssh session, 切换到geth用户环境 `su - geth` , 然后输入： `cat smartContract_example01.sol | tr '\n' ' '` . 切换到以太坊控制台，把合约代码保存为一个变量:

```
var greeterSource = 'contract mortal { address owner; function mortal() { owner = msg.sender; } function kill() { if (msg.sender == owner) selfdestruct(owner); } } contract greeter is mortal { string greeting; function greeter(string _greeting) public { greeting = _greeting; } function greet() constant returns (string) { return greeting; } }'
```

第二步 编译合约代码

然后编译合约代码：

```
var greeterCompiled = web3.eth.compile.solidity(greeterSource)
```

`greeterCompiled.Token.code` 可以看到编译好的二进制代码

`greeterCompiled.Token.info.abiDefinition` 可以看到合约的ABI

第三步 设置希望返回的字符串

```
var _greeting = "Hello World!"
```

第四步 部署合约

接下来我们要把编译好的合约部署到网络上去．

首先我们用ABI来创建一个javascript环境中的合约对象：

```
var greeterContract = web3.eth.contract(greeterCompiled.greeter.info.abiDefinition);
```

我们通过合约对象来部署合约：

```
var greeter = greeterContract.new(_greeting,{from:web3.eth.accounts[0], data: greeterC
ompiled.greeter.code, gas: 300000}, function(e, contract){
    if(!e) {
      if(!contract.address) {
        console.log("Contract transaction send: TransactionHash: " + contract.transact
ionHash + " waiting to be mined...");
      } else {
        console.log("Contract mined! Address: " + contract.address);
        console.log(contract);
      }
    }
})
```

- greeterContract.new方法的第一个参数设置了这个新合约的构造函数初始化的值
- greeterContract.new方法的第二个参数设置了这个新合约的创建者地址from, 这个新合约的代码data, 和用于创建新合约的费用gas．gas是一个估计值，只要比所需要的gas多就可以 ，合约创建完成后剩下的gas会退还给合约创建者．
- greeterContract.new方法的第三个参数设置了一个回调函数，可以告诉我们部署是否成功．

contract.new执行时会提示输入钱包密码．执行成功后，我们的合约Token就已经广播到网络上了． 此时只要等待矿工把我们的合约打包保存到以太坊区块链上，部署就完成了．

第五步 挖矿

在公有链上，矿工打包平均需要15秒，在私有链上，我们需要自己来做这件事情．首先开启挖矿：

```
miner.start(1)
```

此时需要等待一段时间，以太坊节点会生成挖矿必须的数据，这些数据都会放到内存里面．在数据生成好之后，挖矿就会开始，稍后就能在控制台输出中看到类似：

```
...
I0714 22:00:19.694219 ethash.go:291] Generating DAG: 97%
I0714 22:00:22.987934 ethash.go:291] Generating DAG: 98%
I0714 22:00:26.543035 ethash.go:291] Generating DAG: 99%
I0714 22:00:29.912655 ethash.go:291] Generating DAG: 100%
I0714 22:00:29.915580 ethash.go:276] Done generating DAG for epoch 2, it took 5m34.983
289765s
```

第六步 停止挖矿(可选)

当生成DAG结束,提示已经挖出至少一个矿以后,我们需要停止挖矿(当然,你也可以不停,就是会一直输出)

```
miner.stop()
```

第七步 部署在其他节点上

现在,你已经成功部署了一个智能合约,当运行以下代码时:

```
//由于该命令未改变blockchain,所以不会有任何花费
greeter.greet();
```

命令行上会出现如下返回结果:

```
'Hello World!'
```

好了,我们的第一个智能合约程序 "Hello World!" 已经完成了,但是目前它只有一个节点!

第八步 部署在其他节点上

为了使得其他人可以运行你的智能合约，你需要两个信息：

1. 智能合约地址Address
2. 智能合约ABI（Application Binary Interface），ABI其实就是一个有序的用户手册，描述

了所有方法的名字和如何调用它们。 我们可以使用如下代码获得其ABI和智能合约地址:

```
greeterCompiled.greeter.info.abiDefinition;
greeter.address;
```

然后你可以实例化一个JavaScript对象，该对象可以用来在任意联网机器上调用该合约，此处***ABI***和***Address***是上述代码返回值。

```
var greeter = eth.contract(ABI).at(Address);
```

第九步 自毁程序

一个交易需要被发送到网络需要支付费用，自毁程序是对网络的补充，花费的费用远小于一次常用交易。

你可以通过以下代码来检验是否成功，如果自毁程序运行成功以下代码会返回0：

```
greeter.kill.sendTransaction({from:eth.accounts[0]})
```

参考文献

THE GREETER YOUR DIGITAL PAL WHO'S FUN TO BE WITH

以太坊本地私有链开发环境搭建

小结

Hyperledger - 超级账本项目

Uneasy lies the head that wears a crown.

Hyperledger 项目是首个面向企业的开放区块链技术的重要探索。在 Linux 基金会的支持下，吸引了包括 IBM、Intel、摩根等在内的众多科技和金融巨头的参与。

本章将介绍 Hyperledger 项目的历史，并以核心的 fabric 项目为例，讲解如何快速安装部署和应用区块链系统。

简介

历史

区块链已经成为当下最受人关注的开源技术，有人说它将颠覆金融行业的未来。然而对很多人来说，区块链技术难以理解和实现，而且缺乏统一的规范。

2015 年 12 月，开源世界的旗舰——Linux 基金 会牵头，联合 30 家初始企业成员（包括 IBM、Accenture、Intel、J.P.Morgan、R3、DAH、DTCC、FUJITSU、HITACHI、SWIFT、Cisco 等），共同 宣告 了 Hyperledger 项目的成立。该项目试图打造一个透明、公开、去中心化的分布式账本项目，作为区块链技术的开源规范和标准，让更多的应用能更容易的建立在区块链技术之上。项目官方信息网站在 hyperledger.org，

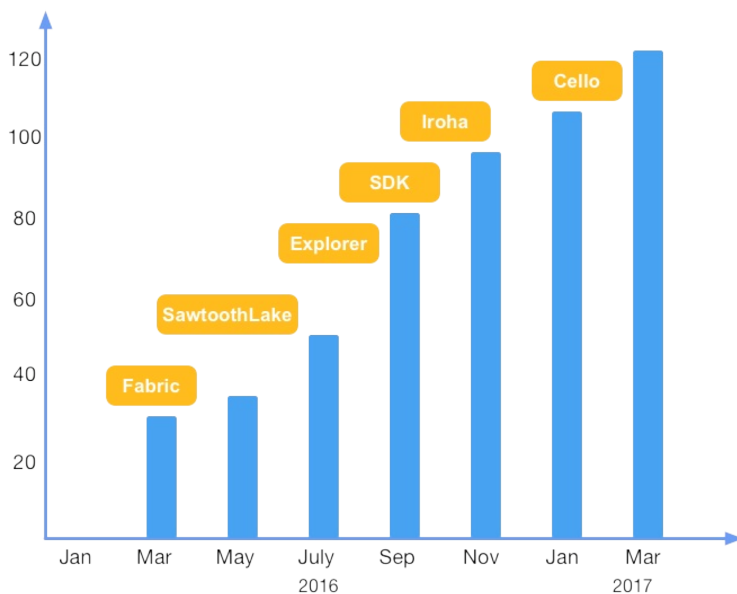


图 *1.11.1.1 - Hyperledger* 项目快速成长

目前已经有超过 120 家全球知名企业和机构（大部分均为各自行业的领导者）宣布加入 Hyperledger 项目，其中包括 30 家来自中国本土的企业，包括艾亿新融旗下的艾亿数融科技公司（2016.05.19）、Onchain（2016.06.22）、比邻共赢（Belink）信息技术有限公司（2016.06.22）、BitSE（2016.06.22）、布比（2016.07.27）、三一重工（2016.08.30）、万达金融（2016.09.08）、华为（2016.10.24）等。

如果说以比特币为代表的货币区块链技术为 1.0，以以太坊为代表的合同区块链技术为 2.0，那么实现了完备的权限控制和安全保障的 Hyperledger 项目毫无疑问代表着 3.0 时代的到来。

IBM 贡献了数万行已有的 Open Blockchain 代码，Digital Asset 则贡献了企业和开发者相关资源，R3 贡献了新的金融交易架构，Intel 也刚贡献了跟分布式账本相关的代码。

Hyperledger 社区由技术委员会（Technical Steering Committee，TSC）指导，首任主席由来自 IBM 开源技术部 CTO 的 Chris Ferris 担任，管理组主席则由来自 Digital Asset Holdings 的 CEO Blythe Masters 担任。另外，自 2016 年 5 月起，Apache 基金会创始人 Brian Behlendorf 担任超级账本项目的首位执行董事。2016 年 12 月，中国技术工作组 正式成立，负责本土社区组织和技术引导工作。官方网站也提供了十分详细的 组织结构信息。

该项目的出现，实际上宣布区块链技术已经不再是仅面向“社会实验”性质的应用场景，它已经正式被主流机构和企业市场认可；同时，Hyperledger 首次提出和实现的完备权限管理、创新的一致性算法和可拔插、可扩展的框架，对于区块链相关技术和产业的发展都将产生深远的影响。

主要项目

代码托管在 Gerrit 和 Github（自动从 gerrit 上同步）上。

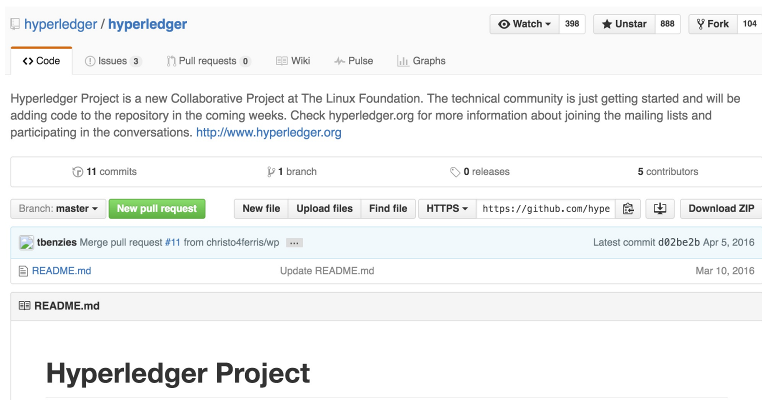


图 *1.11.1.2 - Hyperledger*

目前主要包括三大账本平台项目和若干其它项目。

账本平台项目：

- Fabric：包括 Fabric、Fabric CA、Fabric SDK（包括 Node.Js、Python 和 Java 等语言）和 fabric-api、fabric-sdk-node、fabric-sdk-py 等，目标是区块链的基础核心平台，支持 pbft 等新的 consensus 机制，支持权限管理，最早由 IBM 和 DAH 发起；
- SawToothLake：包括 arcade、core、dev-tools、validator、mktplace 等。是 Intel 主要发起和贡献的区块链平台，支持全新的基于硬件芯片的共识机制 Proof of Elapsed Time（PoET）。
- Iroha：账本平台项目，基于 C++ 实现，带有不少面向 Web 和 Mobile 的特性，主要由 Soramitsu 发起和贡献。

其它项目：

- Blockchain Explorer：提供 Web 操作界面，通过界面快速查看查询绑定区块链的状态（区块个数、交易历史）信息等。
- Cello：提供"Blockchain as a Service" 功能，使用 Cello，管理员可以轻松获取和管理多条区块链；应用开发者可以无需关心如何搭建和维护区块链。

目前，所有项目均处于孵化（Incubation）状态。

项目原则

项目约定共同遵守的 基本原则 为：

- 重视模块化设计，包括交易、合同、一致性、身份、存储等技术场景；
- 代码可读性，保障新功能和模块都可以很容易添加和扩展；
- 演化路线，随着需求的深入和更多的应用场景，不断增加和演化新的项目。

如果你对 Hyperledger 的源码实现感兴趣，可以参考 Hyperledger 源码分析之 Fabric。

社区组织

任何一个成功的开源项目，都离不开一个健康繁荣的社区。

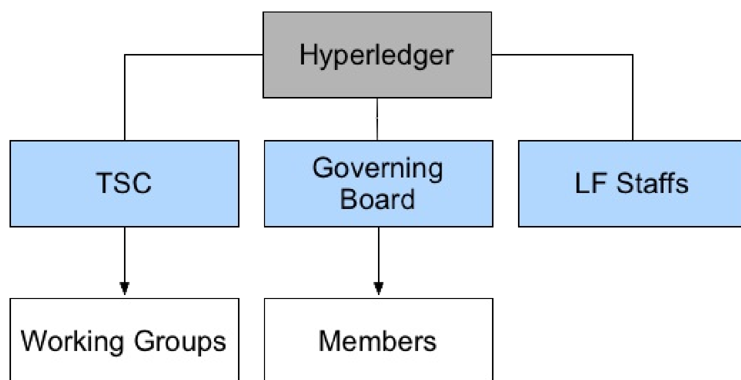


图 *1.11.2.1 - Hyperledger* 社区组织结构

社区目前主要是三驾马车领导的结构：

- Technical Steering Committee （技术委员会）：负责技术相关的工作，下设多个工作组，具体带动各个项目的发展；
- Governing Board （管理董事会）：负责社区组织的整体决策，由 Hyperledger 会员中推选出代表；
- Linux Foundation（Linux 基金会）：协助 Hyperledger 社区在 Linux 基金会的支持下发展。

中国技术工作组

随着开源精神和开源文化在中国的普及，越来越多的企业和组织开始意识到共同构建一个健康的生态系统的重要性，也愿意为开源事业做出一定贡献。

Linux 基金会和 Hyperledger 社区十分重视项目在大中华区的应用和落地情况，并希望能为中国技术人员贡献开源社区提供方便。在此背景下，Hyperledger 执行董事 Brian Behlendorf 于 2016 年 12 月 1 日 提议 成立 中国技术工作组，即 TWG-China，并得到了 TSC 的一致支持通过。

工作组的 主要职责 包括：

- 带领和引导大中华区的技术相关活动，包括贡献代码、指南文档、项目提案等。
- 推动技术相关的交流、会员之间的合作和实践案例的落地；
- 通过邮件列表、rocketchat、论坛等方式提供会员的技术交流；

- 协助举办社区活动，包括 meetup、黑客松、hackfest、技术分享、培训等。

目前，工作组由来自 IBM、万达、华为等 Hyperledger 成员企业的技术专家领导，并得到了来自社区的众多志愿者的支持。

开发和提交代码

安装环境

推荐在 Linux（如 Ubuntu 14.04+）或 MacOS 环境中开发代码，并安装如下工具。

- git：用来获取代码。
- golang 1.6+：安装成功后需要配置 $GOPATH 等环境变量。
- Docker 1.12+：用来支持容器环境，注意 MacOS 下要用 Docker for Mac。

获取代码

首先注册 Linux foundation ID，并登陆 https://gerrit.hyperledger.org/，添加个人 ssh pub key。

查看项目列表，找到对应项目，以 fabric 为例，采用 `Clone with commit-msg hook` 的方式来获取。

典型的，执行如下命令获取代码，放到 `$GOPATH/src/github.com/hyperledger/` 路径下，其中 `LF_ID` 替换为你的 Linux foundation id。

```
$ mkdir $GOPATH/src/github.com/hyperledger/
$ cd $GOPATH/src/github.com/hyperledger/
$ git clone ssh://LF_ID@gerrit.hyperledger.org:29418/fabric && scp -p -P 29418 LF_ID@gerrit.hyperledger.org:hooks/commit-msg fabric/.git/hooks/
```

如果没有添加个人 ssh pubkey，则可以通过 https 方式 clone，需要输入用户名和密码信息。

```
git clone http://LF_ID@gerrit.hyperledger.org/r/fabric && (cd fabric && curl -kLo `git rev-parse --git-dir`/hooks/commit-msg http://LF_ID@gerrit.hyperledger.org/r/tools/hooks/commit-msg; chmod +x `git rev-parse --git-dir`/hooks/commit-msg)
```

编译和测试

大部分编译和安装过程都可以利用 Makefile 来执行，包括如下常见操作。

安装 go tools

执行

```
$ make gotools
```

语法格式检查

执行

```
$ make linter
```

编译 peer

执行

```
$ make peer
```

会自动编译生成 Docker 镜像，并生成本地 peer 可执行文件。

注意：有时候会因为获取安装包不稳定而报错，需要执行 `make clean`，然后再次执行。

生成 Docker 镜像

执行

```
$ make images
```

执行所有的检查和测试

执行

```
$ make checks
```

执行单元测试

执行

```
$ make unit-test
```

如果要运行某个特定单元测试，则可以通过类似如下格式。

```
$ go test -v -run=TestGetFoo
```

执行 BDD 测试

需先生成本地 Docker 镜像。

执行

```
$ make behave
```

提交代码

仍然使用 Linux foundation ID 登录 jira.hyperledger.org，查看有没有未分配的任务，如果对某个任务感兴趣，可以添加自己为 assignee，如对 FAB-XXX 任务。

本地创建新的分支 FAB-XXX。

```
$ git checkout -b FAB-XXX
```

实现任务代码，完成后，执行语法格式检查和测试等，确保所有检查和测试都通过。

提交代码到本地仓库。

```
$ git commit -a -s
```

会打开一个窗口需要填写 commit 信息，格式一般要求为：

```
Simple words to describe main change

This fixes #FAB-XXX.

A more detailed description can be here, with several
paragraphs and sentences...
```

之后使用 git review 命令推送到远端仓库。

```
$ git review
```

提交成功后，可以打开 gerrit.hyperledger.org/r/，查看自己最新提交的 patchset 信息，添加几位 reviewer。之后就是等待开发者团队的 review 结果，如果得到通过，则会被项目的 maintainer 们 merge 到主分支。否则还需要针对大家提出的建议进一步的修正。

修正过程跟提交代码过程类似，唯一不同是提交的时候使用

```
$ git commit -a --amend
```

表示这个提交是对旧提交的一次修订。

完整流程

总结，完整的流程如下图所示。

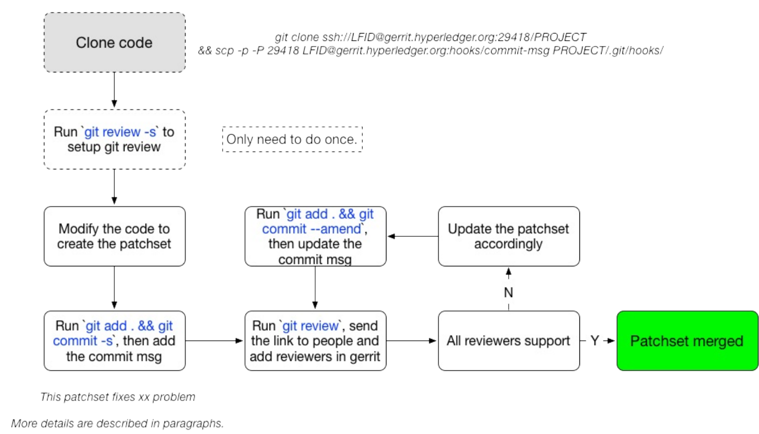


图 *1.11.4.1* - 代码提交流程

小结

Hyperledger 是 Linux 基金会支持的分布式账本平台，这是开源界试图构建一套标准化分布式账本平台的重要尝试。

类似的项目还包括 以太坊平台、R3 CEV 牵头的 Corda 项目、微软的 bletchley 项目 等。

Fabric 部署与管理

简介

使用 1.0 版本

Hyperledger Fabric 1.0 版本整体 重新设计了架构，新的设计可以实现更好的扩展性和安全性。

下载 Compose 模板文件

```
$ git clone https://github.com/yeasy/docker-compose-files
```

进入 `hyperledger/1.0` 目录，查看包括若干模板文件，功能如下。

文件	功能
orderer-base.yaml	orderer 节点的基础服务模板
peer-base.yaml	peer 节点的基础服务模板
docker-compose-base.yaml	包含 orderer 和 peers 组织结构的基础服务模板
docker-compose-1peer.yaml	使用自定义的 `channel` 启动一个最小化的环境，包括 1 个 peer 节点、1 个 orderer 节点、1 个 CA 节点、1 个 cli 节点
docker-compose-2orgs-4peers.yaml	使用自定义的 `channel` 启动一个环境，包括 4 个 peer 节点、1 个 orderer 节点、1 个 CA 节点、1 个 cli 节点
docker-compose-2orgs-4peers-couchdb.yaml	启动一个带有 couchdb 服务的网络环境
docker-compose-2orgs-4peers-event.yaml	启动一个带有 event 事件服务的网络环境
e2e_cli/channel-artifacts	存放创建 orderer, channel, anchor peer 操作时的配置文件
e2e_cli/crypto-config	存放 orderer 和 peer 相关证书
e2e_cli/example	用来测试的 chaincode
scripts/setup_Docker.sh	安装并配置 dokcer 和 docker-compose
scripts/download_images.sh	下载依赖镜像
scripts/start_fabric.sh	快速启动一个fabric 网络
scripts/initialize.sh	自动化测试脚本，用来初始化 `channel` 和 `chaincode`
scripts/test_4peers.sh	自动化测试脚本，用来执行 chaincode 操作
scripts/cleanup_env.sh	容器，镜像自动清除脚本
scripts/test_1peer.sh	测试1个peer网络的自动化脚本
kafka/	基于kafka 的 ordering 服务

安装 Docker 和 docker-compose

docker 及 docker-compose 可以自行手动安装。也可以通过 hyperledger/1.0/scripts 提供的 `setup_Docker.sh` 脚本自动安装。

```
$ bash scripts/setup_Docker.sh
```

获取 Docker 镜像

Docker 镜像可以自行从源码编译（`make docker`），或从 DockerHub 仓库下载。

执行脚本获取

直接执行 hyperledger/1.0/scripts 提供的 `download_images.sh` 脚本获取。

```
$ bash scripts/download_images.sh
```

从官方仓库获取

从社区 DockerHub 仓库下载。

```
# pull fabric images
ARCH=x86_64
BASEIMAGE_RELEASE=0.3.1
BASE_VERSION=1.0.0
PROJECT_VERSION=1.0.0
IMG_TAG=1.0.0

echo "Downloading fabric images from DockerHub...with tag = ${IMG_TAG}... need a while"

# TODO: we may need some checking on pulling result?
docker pull hyperledger/fabric-peer:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-orderer:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-ca:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-tools:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-ccenv:$ARCH-$PROJECT_VERSION
docker pull hyperledger/fabric-baseimage:$ARCH-$BASEIMAGE_RELEASE
docker pull hyperledger/fabric-baseos:$ARCH-$BASEIMAGE_RELEASE

# Only useful for debugging
# docker pull yeasy/hyperledger-fabric

echo "===Re-tagging images to *latest* tag"
docker tag hyperledger/fabric-peer:$ARCH-$IMG_TAG hyperledger/fabric-peer
docker tag hyperledger/fabric-orderer:$ARCH-$IMG_TAG hyperledger/fabric-orderer
docker tag hyperledger/fabric-ca:$ARCH-$IMG_TAG hyperledger/fabric-ca
docker tag hyperledger/fabric-tools:$ARCH-$IMG_TAG hyperledger/fabric-tools
```

从第三方仓库获取

这里也提供了调整（基于 golang:1.8 基础镜像制作）后的第三方镜像，与社区版本功能是一致的。

通过如下命令拉取相关镜像，并更新镜像别名。

```
$ ARCH=x86_64
$ BASEIMAGE_RELEASE=0.3.1
$ BASE_VERSION=1.0.0
$ PROJECT_VERSION=1.0.0
$ IMG_TAG=1.0.0
$ docker pull yeasy/hyperledger-fabric-base:$IMG_VERSION \
  && docker pull yeasy/hyperledger-fabric-peer:$IMG_VERSION \
  && docker pull yeasy/hyperledger-fabric-orderer:$IMG_VERSION \
  && docker pull yeasy/hyperledger-fabric-ca:$IMG_VERSION \
  && docker pull hyperledger/fabric-couchdb:$ARCH-$IMG_VERSION \
  && docker pull hyperledger/fabric-kafka:$ARCH-$IMG_VERSION \
  && docker pull hyperledger/fabric-zookeeper:$ARCH-$IMG_VERSION

$ docker tag yeasy/hyperledger-fabric-peer:$IMG_VERSION hyperledger/fabric-peer \
  && docker tag yeasy/hyperledger-fabric-orderer:$IMG_VERSION hyperledger/fabric-order
er \
  && docker tag yeasy/hyperledger-fabric-ca:$IMG_VERSION hyperledger/fabric-ca \
  && docker tag yeasy/hyperledger-fabric-peer:$IMG_VERSION hyperledger/fabric-tools \
  && docker tag yeasy/hyperledger-fabric-base:$IMG_VERSION hyperledger/fabric-ccenv:$A
RCH-$PROJECT_VERSION \
  && docker tag yeasy/hyperledger-fabric-base:$IMG_VERSION hyperledger/fabric-baseos:$
ARCH-$BASEIMAGE_RELEASE \
  && docker tag yeasy/hyperledger-fabric-base:$IMG_VERSION hyperledger/fabric-baseimag
e:$ARCH-$BASEIMAGE_RELEASE \
  && docker tag hyperledger/fabric-couchdb:$ARCH-$IMG_VERSION hyperledger/fabric-couch
db \
  && docker tag hyperledger/fabric-zookeeper:$ARCH-$IMG_VERSION hyperledger/fabric-zoo
keeper \
  && docker tag hyperledger/fabric-kafka:$ARCH-$IMG_VERSION hyperledger/fabric-kafka
```

启动 fabric 1.0 网络

通过如下命令快速启动。

```
$ bash scripts/start_fabric.sh
```

或者

```
$ docker-compose -f docker-compose-2orgs-4peers.yaml up
```

注意输出日志中无错误信息。

此时，系统中包括 7 个容器。

```
$ docker ps -a
CONTAINER ID        IMAGE                         COMMAND                  CREATED
        STATUS              PORTS
                            NAMES
8683435422ca        hyperledger/fabric-peer       "bash -c 'while true;"   19 seconds a
go      Up 18 seconds       7050-7059/tcp
                            fabric-cli
f284c4dd26a0        hyperledger/fabric-peer       "peer node start --pe"   22 seconds a
go      Up 19 seconds       7050/tcp, 0.0.0.0:7051->7051/tcp, 7052/tcp, 7054-7059/tcp,
 0.0.0.0:7053->7053/tcp     peer0.org1.example.com
95fa3614f82c        hyperledger/fabric-ca         "fabric-ca-server sta"   22 seconds a
go      Up 19 seconds       0.0.0.0:7054->7054/tcp
                            fabric-ca
833ca0d8cf41        hyperledger/fabric-orderer    "orderer"                22 seconds a
go      Up 19 seconds       0.0.0.0:7050->7050/tcp
                            orderer.example.com
cd21cfff8298        hyperledger/fabric-peer       "peer node start --pe"   22 seconds a
go      Up 20 seconds       7050/tcp, 7052/tcp, 7054-7059/tcp, 0.0.0.0:9051->7051/tcp,
 0.0.0.0:9053->7053/tcp     peer0.org2.example.com
372b583b3059        hyperledger/fabric-peer       "peer node start --pe"   22 seconds a
go      Up 20 seconds       7050/tcp, 7052/tcp, 7054-7059/tcp, 0.0.0.0:10051->7051/tcp
, 0.0.0.0:10053->7053/tcp   peer1.org2.example.com
47ce30077276        hyperledger/fabric-peer       "peer node start --pe"   22 seconds a
go      Up 20 seconds       7050/tcp, 7052/tcp, 7054-7059/tcp, 0.0.0.0:8051->7051/tcp,
 0.0.0.0:8053->7053/tcp     peer1.org1.example.com
```

测试网络

启动 fabric 网络后，可以进行 chaincode 操作，验证网络是否启动正常。

进入到 cli 容器里面，执行 `initialize.sh` 和 `test_4peers.sh` 脚本。

通过如下命令进入容器 cli 并执行测试脚本。

```
$ docker exec -it fabric-cli bash
$ bash ./scripts/initialize.sh
```

注意输出日志无错误提示，最终返回结果应该为：

```
UTC [main] main -> INFO 00c Exiting.....
===================== Chaincode Instantiation on PEER2 on channel 'businesschannel' is
 successful =====================


===================== All GOOD, initialization completed =====================


 _____   _   _   ____
| ____| | \ | | |  _ \
|  _|   |  \| | | | | |
| |___  | |\  | | |_| |
|_____| |_| \_| |____/
```

之后同样是在 `cli` 容器里执行 `test_4peers.sh` 脚本

```
$ bash ./scripts/test_4peers.sh
```

输出日志无错误提示，最终返回结果应该为：

```
Query Result: 80
UTC [main] main -> INFO 008 Exiting.....
===================== Query on PEER3 on channel 'businesschannel' is successful ======
===============

===================== All GOOD, End-2-End execution completed =====================


 _____   _   _   ____
| ____| | \ | | |  _ \
|  _|   |  \| | | | | |
| |___  | |\  | | |_| |
|_____| |_| \_| |____/
```

至此，整个网络启动并验证成功。

Fabric v0.6

Fabric 目前的稳定版本为 v0.6，最新的版本 1.0 还在演进中，即将发布。

v0.6 的架构相对简单，适合作为实验或 PoC 场景使用。

Fabric v0.6 安装部署

如果是初次接触 Hyperledger Fabric 项目，推荐采用如下的步骤，基于 Docker-Compose 的一键部署。

官方文档现在也完善了安装部署的步骤，具体可以参考代码 `doc` 目录下内容。

动手前，建议适当了解一些 *Docker 相关知识*。

安装 Docker

Docker 支持 Linux 常见的发行版，如 Redhat/Centos/Ubuntu 等。

```
$ curl -fsSL https://get.docker.com/ | sh
```

以 Ubuntu 14.04 为例，安装成功后，修改 Docker 服务配置（ `/etc/default/docker` 文件）。

```
DOCKER_OPTS="$DOCKER_OPTS -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --api-c
ors-header='*'"
```

重启 Docker 服务。

```
$ sudo service docker restart
```

Ubuntu 16.04 中默认采用了 systemd 管理启动服务，Docker 配置文件在 `/etc/systemd/system/docker.service.d/override.conf` 。

修改后，需要通过如下命令重启 Docker 服务。

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker.service
```

安装 docker-compose

首先，安装 python-pip 软件包。

```
$ sudo aptitude install python-pip
```

安装 docker-compose（推荐为 1.7.0 及以上版本）。

```
$ sudo pip install docker-compose>=1.7.0
```

下载镜像

目前 1.0 代码还没有正式发布，推荐使用 v0.6 分支代码进行测试。

下载相关镜像，并进行配置。

```
$ docker pull yeasy/hyperledger-fabric:0.6-dp \
  && docker pull yeasy/hyperledger-fabric-peer:0.6-dp \
  && docker pull yeasy/hyperledger-fabric-base:0.6-dp \
  && docker pull yeasy/blockchain-explorer:latest \
  && docker tag yeasy/hyperledger-fabric-peer:0.6-dp hyperledger/fabric-peer \
  && docker tag yeasy/hyperledger-fabric-base:0.6-dp hyperledger/fabric-baseimage \
  && docker tag yeasy/hyperledger-fabric:0.6-dp hyperledger/fabric-membersrvc
```

也可以使用 官方仓库 中的镜像。

```
$ docker pull hyperledger/fabric-peer:x86_64-0.6.1-preview \
  && docker pull hyperledger/fabric-membersrvc:x86_64-0.6.1-preview \
  && docker pull yeasy/blockchain-explorer:latest \
  && docker tag hyperledger/fabric-peer:x86_64-0.6.1-preview hyperledger/fabric-peer \
  && docker tag hyperledger/fabric-peer:x86_64-0.6.1-preview hyperledger/fabric-baseim
age \
  && docker tag hyperledger/fabric-membersrvc:x86_64-0.6.1-preview hyperledger/fabric-
membersrvc
```

之后，用户可以选择采用不同的一致性机制，包括 noops、pbft 两类。

使用 noops 模式

noops 默认没有采用 consensus 机制，1 个节点即可，可以用来进行快速测试。

```
$ docker run --name=vp0 \
    --restart=unless-stopped \
    -it \
    -p 7050:7050 \
    -p 7051:7051 \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -e CORE_PEER_ID=vp0 \
    -e CORE_PEER_ADDRESSAUTODETECT=true \
    -e CORE_NOOPS_BLOCK_WAIT=10 \
    hyperledger/fabric-peer:latest peer node start
```

使用 PBFT 模式

PBFT 是经典的分布式一致性算法，也是 hyperledger 目前最推荐的算法，该算法至少需要 4 个节点。

首先，下载 Compose 模板文件。

```
$ git clone https://github.com/yeasy/docker-compose-files
```

进入 `hyperledger/0.6/pbft` 目录，查看包括若干模板文件，功能如下。

- `4-peers.yml`：启动 4 个 PBFT peer 节点。
- `4-peers-with-membersrvc.yml`：启动 4 个 PBFT peer 节点 + 1 个 CA 节点，并启用 CA 功能。
- `4-peers-with-explorer.yml`：启动 4 个 PBFT peer 节点 + 1 个 Blockchain-explorer，可以通过 Web 界面监控集群状态。
- `4-peers-with-membersrvc-explorer.yml`：启动 4 个 PBFT peer 节点 + 1 个 CA 节点 + 1 个 Blockchain-explorer，并启用 CA 功能。

例如，快速启动一个 4 个 PBFT 节点的集群。

```
$ docker-compose -f 4-peers.yml up
```

多物理节点部署

上述方案的典型场景是单物理节点上部署多个 Peer 节点。如果要扩展到多物理节点，需要容器云平台的支持，如 Swarm 等。

当然，用户也可以分别在各个物理节点上通过手动启动容器的方案来实现跨主机组网，每个物理节点作为一个 peer 节点。

首先，以 4 节点下的 PBFT 模式为例，配置 4 台互相连通的物理机，分别按照上述步骤配置 Docker，下载镜像。

4 台物理机分别命名为 vp0 ~ vp3。

vp0

vp0 作为初始的探测节点。

```
$ docker run --name=vp0 \
    --net="host" \
    --restart=unless-stopped \
    -it --rm \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -e CORE_PEER_ID=vp0 \
    -e CORE_PBFT_GENERAL_N=4 \
    -e CORE_LOGGING_LEVEL=debug \
    -e CORE_PEER_ADDRESSAUTODETECT=true \
    -e CORE_PEER_NETWORKID=dev \
    -e CORE_PEER_VALIDATOR_CONSENSUS_PLUGIN=pbft \
    -e CORE_PBFT_GENERAL_MODE=batch \
    -e CORE_PBFT_GENERAL_TIMEOUT_REQUEST=10s \
    hyperledger/fabric-peer:latest peer node start
```

vp1 ~ vp3

以 vp1 为例，假如 vp0 的地址为 10.0.0.1。

```
$ NAME=vp1
$ ROOT_NODE=10.0.0.1
$ docker run --name=${NAME} \
    --net="host" \
    --restart=unless-stopped \
    -it --rm \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -e CORE_PEER_ID=${NAME} \
    -e CORE_PBFT_GENERAL_N=4 \
    -e CORE_LOGGING_LEVEL=debug \
    -e CORE_PEER_ADDRESSAUTODETECT=true \
    -e CORE_PEER_NETWORKID=dev \
    -e CORE_PEER_VALIDATOR_CONSENSUS_PLUGIN=pbft \
    -e CORE_PBFT_GENERAL_MODE=batch \
    -e CORE_PBFT_GENERAL_TIMEOUT_REQUEST=10s \
    -e CORE_PEER_DISCOVERY_ROOTNODE=${ROOT_NODE}:7051 \
    hyperledger/fabric-peer:latest peer node start
```

服务端口

Hyperledger 默认监听的服务端口包括：

- 7050: REST 服务端口，推荐 NVP 节点开放，0.6 之前版本中为 5000；
- 7051：peer gRPC 服务监听端口，0.6 之前版本中为 30303；
- 7052：peer CLI 端口，0.6 之前版本中为 30304；
- 7053：peer 事件服务端口，0.6 之前版本中为 31315；
- 7054：eCAP
- 7055：eCAA
- 7056：tCAP

- 7057：tCAA
- 7058：tlsCAP
- 7059：tlsCAA

使用 chaincode

下面演示 example02 chaincode，完成两方（如 a 和 b）之间进行价值的转移。

部署 chaincode

集群启动后，进入一个 VP 节点。以 pbft 模式为例，节点名称为 `pbft_vp0_1` 。

```
$ docker exec -it pbft_vp0_1 bash
```

部署 chaincode example02 。

```
$ peer chaincode deploy -p github.com/hyperledger/fabric/examples/chaincode/go/chainco
de_example02 -c '{"Function":"init", "Args": ["a","100", "b", "200"]}'
03:08:44.740 [chaincodeCmd] chaincodeDeploy -> INFO 001 Deploy result: type:GOLANG cha
incodeID:<path:"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example0
2" name:"ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5
dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539" > ctorMsg:<args:"init" args:"a" a
rgs:"100" args:"b" args:"200" >
Deploy chaincode: ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb
9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
03:08:44.740 [main] main -> INFO 002 Exiting.....
```

返回 chaincode id 为 `ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539` ，后面将用这个 id 来标识这次交易。为了方便，把它记录到环境变量 CC_ID 中。

```
$ CC_ID="ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5
dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539"
```

部署成功后，系统中会自动生成几个 chaincode 容器，例如

```
CONTAINER ID        IMAGE
                                                                          COMMAND
            CREATED             STATUS              PORTS
      NAMES
e86c26bad76f        dev-vp1-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05
eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539   "/opt/gopath/
bin/ee5b"   2 minutes ago       Up 2 minutes
      dev-vp1-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070
549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
597ebaf929a0        dev-vp2-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05
eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539   "/opt/gopath/
bin/ee5b"   2 minutes ago       Up 2 minutes
      dev-vp2-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070
549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
8748a3b47312        dev-vp3-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05
eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539   "/opt/gopath/
bin/ee5b"   2 minutes ago       Up 2 minutes
      dev-vp3-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070
549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
cf6e762f6a2e        dev-vp0-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05
eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539   "/opt/gopath/
bin/ee5b"   2 minutes ago       Up 2 minutes
      dev-vp0-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070
549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
```

查询 chaincode

查询 a 手头的价值，为初始值 100。

```
$ peer chaincode query -n ${CC_ID} -c '{"Function": "query", "Args": ["a"]}'
03:22:31.420 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Successfully queried tr
ansaction: chaincodeSpec:<type:GOLANG chaincodeID:<name:"ee5b24a1f17c356dd5f6e37307922
e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab181
2590cd0f78539" > ctorMsg:<args:"query" args:"a" > >
Query Result: 100
03:22:31.420 [main] main -> INFO 002 Exiting.....
```

调用 chaincode

a 向 b 转账 10 元。

```
$ peer chaincode invoke -n ${CC_ID} -c '{"Function": "invoke", "Args": ["a", "b", "10"
]}'
03:22:57.345 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Successfully invoked tr
ansaction: chaincodeSpec:<type:GOLANG chaincodeID:<name:"ee5b24a1f17c356dd5f6e37307922
e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab181
2590cd0f78539" > ctorMsg:<args:"invoke" args:"a" args:"b" args:"10" > > (fc298ffb-c763
-4ed0-9da2-072de2ab20b1)
03:22:57.345 [main] main -> INFO 002 Exiting.....
```

查询 a 手头的价值，为新的值 90。

```sh $ peer chaincode query -n ${CC_ID} -c '{"Function": "query", "Args": ["a"]}' 03:23:33.045 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Successfully queried transaction: chaincodeSpec: ctorMsg: > Query Result: 90 03:23:33.045 [main] main -> INFO 002 Exiting..... ···
```

权限管理

权限管理机制是 hyperledger fabric 项目的一大特色。下面给出使用权限管理的一个应用案例。

启动集群

首先下载相关镜像。

```
$ docker pull yeasy/hyperledger:latest
$ docker tag yeasy/hyperledger:latest hyperledger/fabric-baseimage:latest
$ docker pull yeasy/hyperledger-peer:latest
$ docker pull yeasy/hyperledger-membersrvc:latest
```

进入 hyperledger 项目，启动带成员管理的 PBFT 集群。

```
$ git clone https://github.com/yeasy/docker-compose-files
$ cd docker-compose-files/hyperledger/0.6/pbft
$ docker-compose -f 4-peers-with-membersrvc.yml up
```

用户登陆

当启用了权限管理后，首先需要登录，例如以内置账户 jim 账户登录。

登录 vp0，并执行登录命令。

```
$ docker exec -it pbft_vp0_1 bash
# peer network login jim
06:57:13.603 [networkCmd] networkLogin -> INFO 001 CLI client login...
06:57:13.603 [networkCmd] networkLogin -> INFO 002 Local data store for client loginTo
ken: /var/hyperledger/production/client/
Enter password for user 'jim': 6avZQLwcUe9b
06:57:25.022 [networkCmd] networkLogin -> INFO 003 Logging in user 'jim' on CLI interf
ace...
06:57:25.576 [networkCmd] networkLogin -> INFO 004 Storing login token for user 'jim'.
06:57:25.576 [networkCmd] networkLogin -> INFO 005 Login successful for user 'jim'.
06:57:25.576 [main] main -> INFO 006 Exiting.....
```

也可以用 REST 方式：

```
POST  HOST:7050/registrar
```

Request：

```
{
  "enrollId": "jim",
  "enrollSecret": "6avZQLwcUe9b"
}
```

Response：

```
{
    "OK": "User jim is already logged in."
}
```

chaincode 部署

登录之后，chaincode 的部署、调用等操作与之前类似，只是需要通过 -u 选项来指定用户名。

在 vp0 上执行命令：

```
# peer chaincode deploy -u jim -p github.com/hyperledger/fabric/examples/chaincode/go/
chaincode_example02 -c '{"Function":"init", "Args": ["a","100", "b", "200"]}'
06:58:20.099 [chaincodeCmd] getChaincodeSpecification -> INFO 001 Local user 'jim' is
already logged in. Retrieving login token.
06:58:22.178 [chaincodeCmd] chaincodeDeploy -> INFO 002 Deploy result: type:GOLANG cha
incodeID:<path:"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example0
2" name:"ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5
dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539" > ctorMsg:<args:"init" args:"a" a
rgs:"100" args:"b" args:"200" >
Deploy chaincode: ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb
9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
06:58:22.178 [main] main -> INFO 003 Exiting.....
```

记录下返回的 chaincode ID。

```
# CC_ID=ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5d
c31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
```

此时，查询账户值应当为初始值。

```
# peer chaincode query -u jim -n ${CC_ID} -c '{"Function": "query", "Args": ["a"]}'
07:28:39.925 [chaincodeCmd] getChaincodeSpecification -> INFO 001 Local user 'jim' is
already logged in. Retrieving login token.
07:28:40.281 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 002 Successfully queried tr
ansaction: chaincodeSpec:<type:GOLANG chaincodeID:<name:"ee5b24a1f17c356dd5f6e37307922
e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab181
2590cd0f78539" > ctorMsg:<args:"query" args:"a" > secureContext:"jim" >
Query Result: 100
07:28:40.281 [main] main -> INFO 003 Exiting.....
```

也可以通过 REST 方式进行：

```
POST  HOST:7050/chaincode
```

Request：

```
{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": 1,
    "chaincodeID":{
        "path":"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example0
2"
    },
    "ctorMsg": {
        "function":"init",
        "args":["a", "1000", "b", "2000"]
    },
    "secureContext": "jim"
  },
  "id": 1
}
```

Response：

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194
fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539"
    },
    "id": 1
}
```

chaincode 调用

在账户 a、b 之间进行转账 10 元的操作。

```
# peer chaincode invoke -u jim -n ${CC_ID} -c '{"Function": "invoke", "Args": ["a", "b
", "10"]}'
07:29:25.245 [chaincodeCmd] getChaincodeSpecification -> INFO 001 Local user 'jim' is
already logged in. Retrieving login token.
07:29:25.585 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 002 Successfully invoked tr
ansaction: chaincodeSpec:<type:GOLANG chaincodeID:<name:"ee5b24a1f17c356dd5f6e37307922
e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab181
2590cd0f78539" > ctorMsg:<args:"invoke" args:"a" args:"b" args:"10" > secureContext:"j
im" > (f8347e3b-7230-4561-9017-3946756a0bf4)
07:29:25.585 [main] main -> INFO 003 Exiting.....
```

也可以通过 REST 方式进行：

```
POST  HOST:7050/chaincode
```

Request：

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"980d4bb7f69578592e5775a6da86d81a221887817d7164d3e9d4d4df1c981440abf9
a61417eaf8ad6f7fc79893da36de2cf4709131e9af39bca6ebc2e5a1cd9d"
      },
      "ctorMsg": {
         "function":"invoke",
         "args":["a", "b", "100"]
      },
    "secureContext": "jim"
  },
  "id": 3
}
```

Response：

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "66308740-a2c5-4a60-81f1-778dbed49cc3"
    },
    "id": 3
}
```

chaincode 查询

查询 a 账户的余额。

```
# peer chaincode query -u jim -n ${CC_ID} -c '{"Function": "query", "Args": ["a"]}'
07:29:55.844 [chaincodeCmd] getChaincodeSpecification -> INFO 001 Local user 'jim' is
already logged in. Retrieving login token.
07:29:56.198 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 002 Successfully queried tr
ansaction: chaincodeSpec:<type:GOLANG chaincodeID:<name:"ee5b24a1f17c356dd5f6e37307922
e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab181
2590cd0f78539" > ctorMsg:<args:"query" args:"a" > secureContext:"jim" >
Query Result: 90
07:29:56.198 [main] main -> INFO 003 Exiting.....
```

也可以通过 REST 方式进行：

```
POST  HOST:7050/chaincode
```

Request：

```
{
  "jsonrpc": "2.0",
  "method": "query",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"980d4bb7f69578592e5775a6da86d81a221887817d7164d3e9d4d4df1c981440abf9
a61417eaf8ad6f7fc79893da36de2cf4709131e9af39bca6ebc2e5a1cd9d"
      },
      "ctorMsg": {
         "function":"query",
         "args":["a"]
      },
      "secureContext": "jim"
  },
  "id": 5
}
```

Response：

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "900"
    },
    "id": 5
}
```

区块信息查询

URL：

```
GET  HOST:7050/chain/blocks/2
```

Response：

```
{
    "transactions": [
        {
            "type": 2,
            "chaincodeID": "EoABMjhiYjJiMjMxNjE3MWE3MDZiYjI4MTBlYzM1ZDA5NWY0MzA4NzdiZj
Q0M2YxMDYxZWYwZjYwYmJlNzUzZWQ0NDA3MDBhNTMxMmMxNjM5MGQzYjMwMTk5ZmU5NDY1YzNiNzVkNTk0NDM1
OGNhYWUwMWNhODFlZjI4MTI4YTFiZmI=",
            "payload": "Cp0BCAESgwESgAEyOGJiMmIyMzE2MTcxYTcwNmJiMjgxMGVjMzVkMDk1ZjQzMD
g3N2JmNDQzZjEwNjFlZjBmNjBiYmU3NTNlZDQ0MDcwMGE1MzEyYzE2MzkwZDNiMzAxOTlmZTk0NjVjM2I3NWQ1
OTQ0MzU4Y2FhZTAxY2E4MWVmMjgxMjhhMWJmYhoTCgZpbnZva2USAWESAWISAzEwMA==",
            "uuid": "2b3b6cf3-9887-4dd5-8f2e-3634ec9c719a",
            "timestamp": {
                "seconds": 1466577447,
                "nanos": 399637431
            },
            "nonce": "5AeA6S1odhPIDiGjFTFG8ttcihOoNNsh",
            "cert": "MIICPzCCAeSgAwIBAgIRAMndnS+Me0G6gs4J9/fb8HcwCgYIKoZIzj0EAwMwMTELM
AkGA1UEBhMCVVMxFDASBgNVBAoTC0h5cGVybGVkZ2VyMQwwCgYDVQQDEwN0Y2EwHhcNMTYwNjIyMDYzMzE4Whc
NMTYwOTIwMDYzMzE4WjAxMQswCQYDVQQGEwJVUzEUMBIGA1UEChMLSHlwZXJsZWRnZXIxDDAKBgNVBAMTA2ppb
TBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABDLd2W8PxzgB4A85Re2x44BApbOGqP05tnkygbXSctLiqi5HVfw
RAACS6znVA9+toni59Yy+XAH3w2offdjFW3mjgdwwgdkwDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/wQCMAAwD
QYDVR0OBAYEBAECAwQwDwYDVR0jBAgwBoAEAQIDBDBNBgYqAwQFBgcBAf8EQAfASTE6bZ0P5mrEzTa5r1UyKFv
+dKezBiGU0V3l2iWzk9evlGMvaC2pwhEKfKDdKxs7YSMYe/7cLq/oF++GBVowSgYGKgMEBQYIBEBEO3TKXuORl
5Geuco8Gnn5TkoIl4+b96aPGDGvKbmDjMXR9vEBuUXTnsbDL53j7kC8/XQs1kZboC1ojLeUSN03MAoGCCqGSM4
9BAMDA0kAMEYCIQCZqyANMFcu1WiMe2So0pC7eRU95F0+qUXLAKZsPWv/YQIhALmNaglP7CoMOe2qxehucmffD
lu0BRLSYDHyV9xcxmkH",
            "signature": "MEYCIQDob3NqdrfwlSGhi+zz+Ypl7S9QQ07RIFr8nV92e8KDNgIhANIljz4t
RS8vwQk01hTemNQFJX2zMI6DhSUFZivbbtoR"
        }
    ],
    "stateHash": "7YUoVvYnMLHbLf47uTixLtkjF6xM9DuvgSWC92MbOUzk09xhcRBBLZqe5FvJElgZemEL
BOcuIFnubL0LiGH0yw==",
    "previousBlockHash": "On4BlpqCYNpugUKluqvOcbvkr3TAQxmlISLdd6qrONtIgmQ4iUDeWxAA9lUC
ceZfF8tke8A0Wy7m9tksNpKodw==",
    "consensusMetadata": "CAI=",
    "nonHashData": {
        "localLedgerCommitTimestamp": {
            "seconds": 1466577447,
            "nanos": 653618964
        },
        "transactionResults": [
            {
                "uuid": "2b3b6cf3-9887-4dd5-8f2e-3634ec9c719a"
            }
        ]
    }
}
```

Python 客户端

前面应用案例，都是直接通过 HTTP API 来跟 hyperledger 进行交互，操作比较麻烦。

还可以直接通过 hyperledger-py 客户端来进行更方便的操作。

安装

```
$ pip install hyperledger --upgrade
```

或直接源码安装

```
$ git clone https://github.com/yeasy/hyperledger-py.git
$ cd hyperledger-py
$ pip install -r requirements.txt
$ python setup.py install
```

使用

```
>>> from hyperledger.client import Client
>>> c = Client(base_url="http://127.0.0.1:7050")
>>> c.peer_list()
{u'peers': [{u'type': 1, u'ID': {u'name': u'vp1'}, u'address': u'172.17.0.2:30303'}, {u
'type': 1, u'ID': {u'name': u'vp2'}, u'address': u'172.17.0.3:30303'}]}
```

更多使用方法，可以参考 API 文档。

其它客户端

目前，HyperLedger Fabric 已经成立了 SDK 工作组。

目前在实现的客户端 SDK 包括：

- Python SDK
- Nodejs SDK

本章小结

区块链应用开发

简介

链码示例一：信息公证

简介

chaincode_example01.go 主要实现如下的功能：

- 初始化，以键值形式存放信息；
- 允许读取和修改键值。

代码中，首先初始化了 `hello_world` 的值，并根据请求中的参数创建修改查询链上 `key` 中的值，本质上实现了一个简单的可修改的键值数据库。

主要函数

- `read` ：读取key `args[0]` 的 value；
- `write` ：创建或修改 key `args[0]` 的 value；
- `init` ：初始化 key `hello_world` 的 value；
- `invoke` ：根据传递参数类型调用执行相应的 `init` 和 `write` 函数；
- `query` ：调用 `read` 函数查询 `args[0]` 的 value。

代码运行分析

`main` 函数作为程序的入口，调用 shim 包的 start 函数，启动 chaincode 引导程序的入口节点。如果报错，则返回。

```
func main() {
    err := shim.Start(new(SimpleChaincode))
    if err != nil {
        fmt.Printf("Error starting Simple chaincode: %s", err)
    }
}
```

当智能合约部署在区块链上，可以通过 rest api 进行交互。

三个主要的函数是 `init` ， `invoke` ， `query` 。在三个函数中，通过 `stub.PutState` 与 `stub.GetState` 存储访问 ledger 上的键值对。

通过 REST API 操作智能合约

假设以 jim 身份登录 pbft 集群，请求部署该 chaincode 的 json 请求格式为：

```
{
    "jsonrpc": "2.0",
    "method": "deploy",
    "params": {
        "type": 1,
        "chaincodeID": {
            "path": "https://github.com/ibm-blockchain/learn-chaincode/finished"
        },
        "ctorMsg": {
            "function": "init",
            "args": [
                "hi there"
            ]
        },
        "secureContext": "jim"
    },
    "id": 1
}
```

目前 path 仅支持 github 上的目录，ctorMsg 中为函数 `init` 的传参。

调用 invoke 函数的 json 格式为：

```
{
    "jsonrpc": "2.0",
    "method": "invoke",
    "params": {
        "type": 1,
        "chaincodeID": {
            "name": "4251b5512bad70bcd0947809b163bbc8398924b29d4a37554f2dc2b033617c19c
c0611365eb4322cf309b9a5a78a5dba8a5a09baa110ed2d8aeee186c6e94431"
        },
        "ctorMsg": {
            "function": "init",
            "args": [
                "swb"
            ]
        },
        "secureContext": "jim"
    },
    "id": 2
}
```

其中 name 字段为 `deploy` 后返回的 message 字段中的字符串。

`query` 的接口也是类似的。

链码示例二：交易资产

简介

chaincode_example02.go 主要实现如下的功能：

- 初始化 A、B 两个账户，并为两个账户赋初始资产值；
- 在 A、B 两个账户之间进行资产交易；
- 分别查询 A、B 两个账户上的余额，确认交易成功；
- 删除账户。

主要函数

- `init` ：初始化 A、B 两个账户；
- `invoke` ：实现 A、B 账户间的转账；
- `query` ：查询 A、B 账户上的余额；
- `delete` ：删除账户。

依赖的包

```
import (
    "errors"
    "fmt"
    "strconv"

    "github.com/hyperledger/fabric/core/chaincode/shim"
)
```

`strconv` 实现 int 与 string 类型之间的转换。

在invoke 函数中，存在：

```
X, err = strconv.Atoi(args[2])
    Aval = Aval - X
    Bval = Bval + X
```

当 `args[2]<0` 时，A 账户余额增加，否则 B 账户余额减少。

可扩展功能

实例中未包含新增账户并初始化的功能。开发者可以根据自己的业务模型进行添加。

数字货币发行与管理

简介

该智能合约实现一个简单的商业应用案例，即数字货币的发行与转账。在这之中一共分为三种角色：中央银行，商业银行，企业。其中中央银行可以发行一定数量的货币，企业之间可以进行相互的转账。主要实现如下的功能：

- 初始化中央银行及其发行的货币数量
- 新增商业银行，同时央行并向其发行一定数量的货币
- 新增企业
- 商业银行向企业转给一定数量的数字货币
- 企业之间进行相互的转账
- 查询企业、银行、交易信息

主要函数

- `init` ：初始化中央银行，并发行一定数量的货币；
- `invoke` ：调用合约内部的函数；
- `query` ：查询相关的信息；
- `createBank` ：新增商业银行，同时央行向其发行一定数量的货币；
- `createCompany` ：新增企业；
- `issueCoin` ：央行再次发行一定数量的货币（归于交易）；
- `issueCoinToBank` ：央行向商业银行转一定数量的数字货币（归于交易）；
- `issueCoinToCp` ：商业银行向企业转一定数量的数字货币（归于交易行为）；
- `transfer` ：企业之间进行相互转账（归于交易行为）；
- `getCompanys` ：获取所有的公司信息，如果企业个数大于10，先访问前10个；
- `getBanks` ：获取所有的商业银行信息，如果商业银行个数大于10，先访问前 10 个
- `getTransactions` ：获取所有的交易记录 如果交易个数大于10，先访问前 10 个；
- `getCompanyById` ：获取某家公司信息；
- `getBankById` ：获取某家银行信息；
- `getTransactionBy` ：获取某笔交易记录；
- `writeCenterBank` ：修改央行信息；
- `writeBank` ：修改商业银行信息；
- `writeCompany` ：修改企业信息；
- `writeTransaction` ：写入交易信息。

数据结构设计

- centerBank 中央银行
  - Name：名称
  - TotalNumber：发行货币总数额
  - RestNumber：账户余额
  - ID：ID固定为 0
- bank 商业银行
  - Name：名称
  - TotalNumber：收到货币总数额
  - RestNumber：账户余额
  - ID：银行 ID
- company 企业
  - Name：名称
  - Number：账户余额
  - ID：企业 ID
- transaction 交易内容
  - FromType：发送角色 //centerBank:0,Bank:1,Company:2
  - FromID：发送方 ID
  - ToType：接收方角色 //Bank:1,Company:2
  - ToID：接收方 ID
  - Time：交易时间
  - Number：交易数额
  - ID：交易 ID

接口设计

`init`

request 参数:

```
args[0] 银行名称
args[1] 初始化发布金额
```

response 参数:

```
{"Name":"XXX","TotalNumber":"0","RestNumber":"0","ID":"XX"}
```

`createBank`

request 参数:

```
args[0] 银行名称
```

response 参数:

```
{"Name":"XXX","TotalNumber":"0","RestNumber":"0","ID":"XX"}
```

`createCompany`

request 参数:

```
args[0] 公司名称
```

response 参数:

```
{"Name":"XXX","Number":"0","ID":"XX"}
```

`issueCoin`

request 参数:

```
args[0] 再次发行货币数额
```

response 参数:

```
{"FromType":"0","FromID":"0","ToType":"0","ToID":"0","Time":"XX","Number":"XX","ID":"XX"}
```

`issueCoinToBank`

request 参数:

```
args[0] 商业银行ID
args[1] 转账数额
```

response 参数:

```
{"FromType":"0","FromID":"0","ToType":"1","ToID":"XX","Time":"XX","Number":"XX","ID":"XX"}
```

`issueCoinToCp`

request 参数:

```
args[0] 商业银行ID
args[1] 企业ID
args[2] 转账数额
```

response 参数:

```
{"FromType":"1","FromID":"XX","ToType":"2","ToID":"XX","Time":"XX","Number":"XX","ID":
"XX"}
```

`transfer`

request 参数:

```
args[0] 转账用户ID
args[1] 被转账用户ID
args[2] 转账余额
```

response 参数:

```
{"FromType":"2","FromID":"XX","ToType":"2","ToID":"XX","Time":"XX","Number":"XX","ID":
"XX"}
```

`getBanks`

response 参数

```
[{"Name":"XXX","Number":"XX","ID":"XX"},{"Name":"XXX","Number":"XX","ID":"XX"},...]
```

`getCompanys`

response 参数

```
[{"Name":"XXX","TotalNumber":"XX","RestNumber":"XX","ID":"XX"},{"Name":"XXX","TotalNum
ber":"XX","RestNumber":"XX","ID":"XX"},...]
```

`getTransactions`

response 参数

```
[{"FromType":"XX","FromID":"XX","ToType":"XX","ToID":"XX","Time":"XX","Number":"XX","ID":"XX"},{"FromType":"XX","FromID":"XX","ToType":"XX","ToID":"XX","Time":"XX","Number":"XX","ID":"XX"},...]
```

getCenterBank

response 参数

```
[{"Name":"XX","TotalNumber":"XX","RestNumber":"XX","ID":"XX"}]
```

getBankById

request 参数

```
args[0] 商业银行ID
```

response 参数

```
[{"Name":"XX","TotalNumber":"XX","RestNumber":"XX","ID":"XX"}]
```

getCompanyById

request 参数

```
args[0] 企业ID
```

response 参数

```
[{"Name":"XXX","Number":"XX","ID":"XX"}]
```

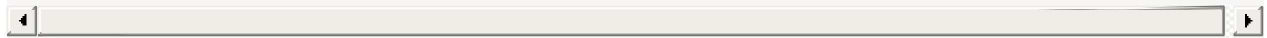
getTransactionById

request 参数

```
args[0] 交易ID
```

response 参数

```
{"FromType":"XX","FromID":"XX","ToType":"XX","ToID":"XX","Time":"XX","Number":"XX","ID"
:"XX"}
```

`writeCenterBank`

request 参数

```
CenterBank
```

response 参数

```
err  nil 为成功
```

`writeBank`

request 参数

```
Bank
```

response 参数

```
err  nil 为成功
```

`writeCompany`

request 参数

```
Company
```

response 参数

```
err  nil 为成功
```

`writeTransaction`

request 参数

```
Transaction
```

response 参数

``` err nil 为成功 ···

# 其它

查询时为了兼顾读速率，将一些信息备份存放在非区块链数据库上也是一个较好的选择。
```

学历认证

功能描述

该 智能合约 实现了一个简单的征信管理的案例。针对于学历认证领域，由于条约公开，在条约外无法随意篡改的特性，天然具备稳定性和中立性。

该智能合约中三种角色如下：

- 学校
- 个人
- 需要学历认证的机构或公司

学校可以根据相关信息在区块链上为某位个人授予学历，相关机构可以查询某人的学历信息，由于使用私钥签名，确保了信息的真实有效。 为了简单，尽量简化相关的业务，另未完成学业的学生因违纪或外出创业退学，学校可以修改其相应的学历信息。

账户私钥应该由安装在本地的客户端生成，本例中为了简便，使用模拟私钥和公钥。

数据结构设计

- 学校
  - 名称
  - 所在位置
  - 账号地址
  - 账号公钥
  - 账户私钥
  - 学校学生
- 个人
  - 姓名
  - 账号地址
  - 过往学历
- 学历信息
  - 学历信息编号
  - 就读学校
  - 就读年份
  - 完成就读年份
  - 就读状态 // 0:毕业 1：退学
- 修改记录（入学也相当于一种修改记录）
  - 编号
  - 学校账户地址（一般根据账户地址可以算出公钥地址，然后可以进行校验）

- 学校签名
- 个人账户地址
- 个人公钥地址（个人不需要公钥地址）
- 修改时间
- 修改操作// 0:正常毕业 1：退学 2:入学

对学历操作信息所有的操作都归为记录。

function及各自实现的功能

- `init` 初始化函数
- `invoke` 调用合约内部的函数
- `updateDiploma` 由学校更新学生学历信息，并签名（返回记录信息）
- `enrollStudent` 学校招生（返回学校信息）
- `createSchool` 添加一名新学校
- `createStudent` 添加一名新学生
- `getStudentByAddress` 通过学生的账号地址访问学生的学历信息
- `getRecordById` 通过Id获取记录
- `getRecords` 获取全部记录（如果记录数大于 10，返回前 10 个）
- `getSchoolByAddress` 通过学校账号地址获取学校的信息
- `getBackgroundById` 通过学历 Id 获取所存储的学历信息
- `writeRecord` 写入记录
- `writeSchool` 写入新创建的学校
- `writeStudent` 写入新创建的学生

接口设计

`createSchool`

request参数:

```
args[0] 学校名称
args[1] 学校所在位置
```

response参数:

```
学校信息的字节数组，当创建一所新学校时，该学校学生账户地址列表为空
```

`createStudent`

request参数：

```
args[0] 学生的姓名
```

response参数：

```
学生信息的字节数组表示，刚创建过往学历信息列表为空
```

`updateDiploma`

request参数

```
args[0] 学校账户地址
args[1] 学校签名
args[2] 待修改学生的账户地址
args[3] //对该学生的学历进行怎样的修改，0：正常毕业  1：退学
```

response参数

```
返回修改记录的字节数组表示
```

`enrollStudent`

request参数:

```
args[0] 学校账户地址
args[1] 学校签名
args[2] 学生账户地址
```

response参数

```
返回修改记录的字节数组表示
```

`getStudentByAddress`

request参数

```
args[0] address
```

response参数

```
学生信息的字节数组表示
```

`getRecordById`

request参数

```
args[0] 修改记录的ID
```

response参数

```
修改记录的字节数组表示
```

`getRecords`

response参数

```
获取修改记录数组（如果个数大于10，返回前10个）
```

`getSchoolByAddress`

request参数

```
args[0] address
```

response参数

```
学校信息的字节数组表示
```

`getBackgroundById`

request参数

```
args[0] ID
```

response参数

```
学历信息的字节数组表示
```

测试

社区能源共享

功能描述

本 合约 以纽约实验性的能源微电网为例，作为一个简单的案例进行实现。

> “在总统大道的一边，五户家庭通过太阳能板发电；在街道的另一边的五户家庭可以购买对面家庭不需要的电力。而连接这项交易的就是区块链网络，几乎不需要人员参与就可以管理记录交易。”但是这个想法是非常有潜力的，能够代表未来社区管理能源系统。”

布鲁克林微电网开发商 LO3 创始人 Lawrence Orsini 说：

> “我们正在这条街道上建立一个可再生电力市场，来测试人们对于购买彼此手中的电力是否感兴趣。如果你在很远的地方生产能源，运输途中会有很多损耗，你也得不到这电力价值。但是如果你就在街对面，你就能高效的利用能源。”

在某一块区域内存在一个能源微电网，每一户家庭可能为生产者也可能为消费者。部分家庭拥有太阳能电池板，太阳能电池板的剩余电量为可以售出的电力的值，为了简化，单位为1.需要电力的家庭可以向有足够余额的电力的家庭购买电力。

账户私钥应该由安装在本地的客户端生成，本例中为了简便，使用模拟私钥和公钥。每位用户的私钥为guid+“1”，公钥为guid+“2”。签名方式简化为私钥+"1"

数据结构设计

在该智能合约中暂时只有一种角色，为每一户家庭用户。

- 家庭用户
  - 账户地址
  - 剩余能量 //部分家庭没有太阳能电池板，值为0
  - 账户余额（电子货币）
  - 编号
  - 状态 //0：不可购买， 1：可以购买
  - 账户公钥
  - 账户私钥
- 交易(一笔交易必须同时具有卖方和买方的公钥签名，方能承认这笔交易。公钥签名生成规则，公钥+待创建交易的ID号，在本交易类型中，只要买家有足够的货币，卖家自动会对交易进行签名)
  - 购买方地址
  - 销售方地址
  - 电量销售量
  - 电量交易金额

  - 编号
  - 交易时间

function及各自实现的功能

- `init` 初始化操作
- `invoke` 调用合约内部的函数
- `query` 查询相关的信息
- `createUser` 创建新用户，并加入到能源微网中 invoke
- `buyByAddress` 向某一位用户购买一定量的电力 invoke
- `getTransactionById` 通过id获取交易内容 query
- `getTransactions` 获取交易（如果交易数大于10，获取前10个） query
- `getHomes` 获取用户（如果用户数大于10，获取前10个） query
- `getHomeByAddress` 通过地址获取用户 query
- `changeStatus` 某一位用户修改自身的状态 invoke
- `writeUser` 将新用户写入到键值对中
- `writeTransaction` 记录交易

接口设计

`createUser`

request参数:

```
args[0] 剩余能量值
args[1] 剩余金额
```

response参数:

```
新建家庭用户的json表示
```

`buyByAddress`

request参数:

```
args[0] 卖家的账户地址
args[1] 买家签名
args[2] 买家的账户地址
args[3] 想要购买的电量数值
```

response参数:

```
购买成功的话返回该transaction的json串。
购买失败返回error
```

`getTransactionById`

request参数:

```
args[0] 交易编号
```

response参数:

```
查询结果的transaction 交易表示
```

`getTransactions`

request参数:

```
none
```

response参数:

```
获取所有的交易列表（如果交易大于10，则返回前10个）
```

`getHomeByAddress`

request参数

```
args[0] address
```

response参数

```
用户信息的json表示
```

`getHomes`

response参数

```
获取所有的用户列表（如果用户个数大于10，则返回前10个）
```

`changeStatus`

request参数:

```
args[0] 账户地址
args[1]  账户签名
args[2] 对自己的账户进行的操作，0：设置为不可购买  1：设置状态为可购买
```

response参数:

```
修改后的用户信息json表示
```

测试

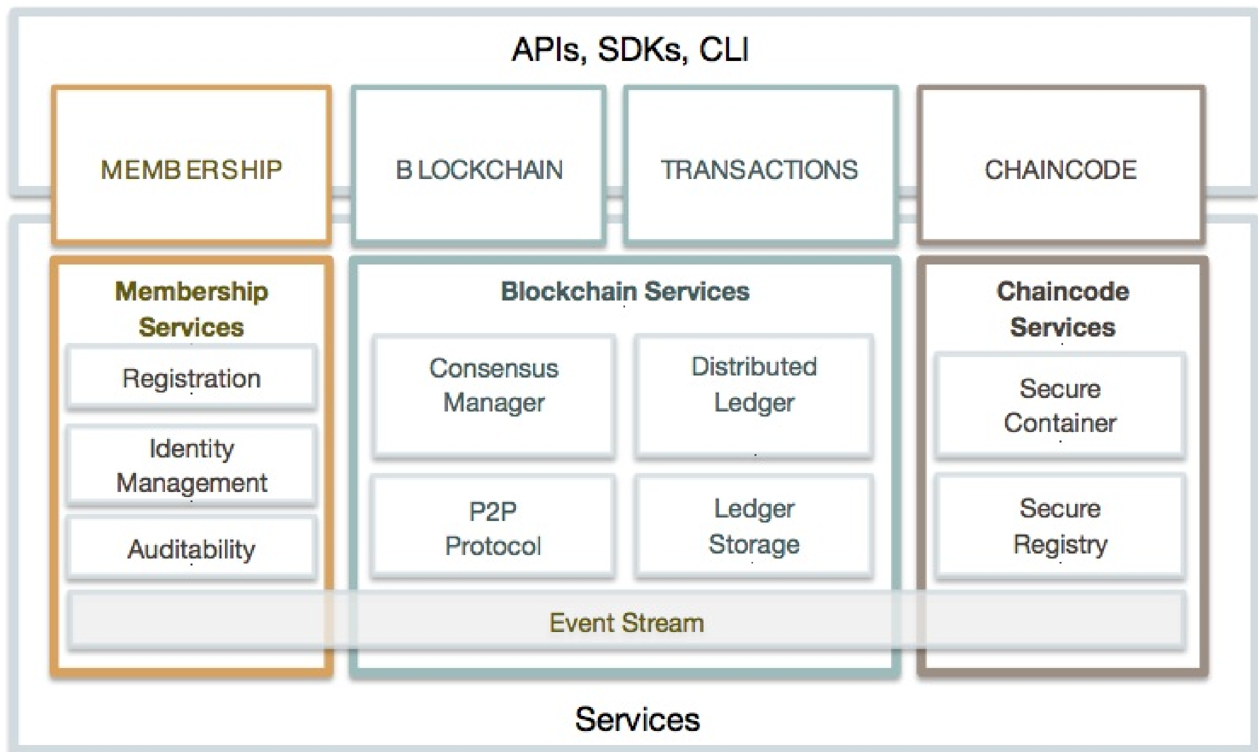
本章小结

Fabric 架构与设计

简介

架构设计

整个功能架构如下图所示。



包括三大组件：区块链服务（Blockchain）、链码服务（Chaincode）、成员权限管理（Membership）。

概念术语

- Auditability（审计性）：在一定权限和许可下，可以对链上的交易进行审计和检查。
- Block（区块）：代表一批得到确认的交易信息的整体，准备被共识加入到区块链中。
- Blockchain（区块链）：由多个区块链接而成的链表结构，除了首个区块，每个区块都包括前继区块内容的 hash 值。
- Certificate Authority（CA）：负责身份权限管理，又叫 Member Service 或 Identity Service。
- Chaincode（链上代码或链码）：区块链上的应用代码，扩展自“智能合约”概念，支持 golang、nodejs 等，运行在隔离的容器环境中。
- Committer（提交节点）：1.0 架构中一种 peer 节点角色，负责对 orderer 排序后的交易进行检查，选择合法的交易执行并写入存储。
- Confidentiality（保密）：只有交易相关方可以看到交易内容，其它人未经授权则无法看到。
- Endorser（背书节点）：1.0 架构中一种 peer 节点角色，负责检验某个交易是否合法，是否愿意为之背书、签名。

- Enrollment Certificate Authority（ECA，注册 CA）：负责成员身份相关证书管理的 CA。
- Ledger（账本）：包括区块链结构（带有所有的可验证交易信息，但只有最终成功的交易会改变世界观）和当前的世界观（world state）。Ledger 仅存在于 Peer 节点。
- MSP（Member Service Provider，成员服务提供者）：成员服务的抽象访问接口，实现对不同成员服务的可拔插支持。
- Non-validating Peer（非验证节点）：不参与账本维护，仅作为交易代理响应客户端的 REST 请求，并对交易进行一些基本的有效性检查，之后转发给验证节点。
- Orderer（排序节点）：1.0 架构中的共识服务角色，负责排序看到的交易，提供全局确认的顺序。
- Permissioned Ledger（带权限的账本）：网络中所有节点必须是经过许可的，非许可过的节点则无法加入网络。
- Privacy（隐私保护）：交易员可以隐藏交易的身份，其它成员在无特殊权限的情况下，只能对交易进行验证，而无法获知身份信息。
- Transaction（交易）：执行账本上的某个函数调用。具体函数在 chaincode 中实现。
- Transactor（交易者）：发起交易调用的客户端。
- Transaction Certificate Authority（TCA，交易 CA）：负责维护交易相关证书管理的 CA。
- Validating Peer（验证节点）：维护账本的核心节点，参与一致性维护、对交易的验证和执行。
- World State（世界观）：是一个键值数据库，chaincode 用它来存储交易相关的状态。

区块链服务

区块链服务提供一个分布式账本平台。一般地，多个交易被打包进区块中，多个区块构成一条区块链。区块链代表的是账本状态机发生变更的历史过程。

交易

交易意味着围绕着某个链码进行操作。

交易可以改变世界状态。

交易中包括的内容主要有：

- 交易类型：目前包括 Deploy、Invoke、Query、Terminate 四种；
- uuid：代表交易的唯一编号；
- 链码编号 chaincodeID：交易针对的链码；
- 负载内容的 hash 值：Deploy 或 Invoke 时候可以指定负载内容；
- 交易的保密等级 ConfidentialityLevel；
- 交易相关的 metadata 信息；
- 临时生成值 nonce：跟安全机制相关；

- 交易者的证书信息 cert；
- 签名信息 signature；
- metadata 信息；
- 时间戳 timestamp。

交易的数据结构（Protobuf 格式）定义为

```
message Transaction {
    enum Type {
        UNDEFINED = 0;
        // deploy a chaincode to the network and call `Init` function
        CHAINCODE_DEPLOY = 1;
        // call a chaincode `Invoke` function as a transaction
        CHAINCODE_INVOKE = 2;
        // call a chaincode `query` function
        CHAINCODE_QUERY = 3;
        // terminate a chaincode; not implemented yet
        CHAINCODE_TERMINATE = 4;
    }
    Type type = 1;
    //store ChaincodeID as bytes so its encrypted value can be stored
    bytes chaincodeID = 2;
    bytes payload = 3;
    bytes metadata = 4;
    string uuid = 5;
    google.protobuf.Timestamp timestamp = 6;

    ConfidentialityLevel confidentialityLevel = 7;
    string confidentialityProtocolVersion = 8;
    bytes nonce = 9;

    bytes toValidators = 10;
    bytes cert = 11;
    bytes signature = 12;
}
```

在 1.0 架构中，一个 transaction 包括如下信息：

[ledger] [channel], **proposal:**[chaincode,] **endorsement:**[proposal hash, simulation result, signature]

- endorsements: proposal hash, simulation result, signature
- function-spec: function name, arguments
- proposal: [channel,] chaincode,

区块

区块打包交易，确认交易后的世界状态。

一个区块中包括的内容主要有：

- 版本号 version：协议的版本信息；
- 时间戳 timestamp：由区块提议者设定；
- 交易信息的默克尔树的根 hash 值：由区块所包括的交易构成；
- 世界观的默克尔树的根 hash 值：由交易发生后整个世界的状态值构成；
- 前一个区块的 hash 值：构成链所必须；
- 共识相关的元数据：可选值；
- 非 hash 数据：不参与 hash 过程，各个 peer 上的值可能不同，例如本地提交时间、交易处理的返回值等；

注意具体的交易信息并不存放在区块中。

区块的数据结构（Protobuf 格式）定义为

```
message Block {
    uint32 version = 1;
    google.protobuf.Timestamp timestamp = 2;
    repeated Transaction transactions = 3;
    bytes stateHash = 4;
    bytes previousBlockHash = 5;
    bytes consensusMetadata = 6;
    NonHashData nonHashData = 7;
}
```

一个真实的区块内容示例：

```
{
    "nonHashData": {
        "localLedgerCommitTimestamp": {
            "nanos": 975295157,
                "seconds": 1466057539
        },
            "transactionResults": [
            {
                "uuid": "7be1529ee16969baf9f3156247a0ee8e7eee99a6a0a816776acff65e6e1de
f71249f4cb1cad5e0f0b60b25dd2a6975efb282741c0e1ecc53fa8c10a9aaa31137"
            }
            ]
    },
        "previousBlockHash": "RrndKwuojRMjOz/rdD7rJD/NUupiuBuCtQwnZG7Vdi/XXcTd2MDyAMsF
AZ1ntZL2/IIcSUeatIZAKS6ss7fEvg==",
        "stateHash": "TiIwROg48Z4xXFFIPEunNpavMxnvmZKg+yFxKK3VBY0zqiK3L0QQ5ILIV85iy7U+
EiVhwEbkBb1Kb7w1ddqU5g==",
        "transactions": [
        {
            "chaincodeID": "CkdnaXRodWIuY29tL2h5cGVybGVkZ2VyL2ZhYnJpYy9leGFtcGxlcy9jaG
FpbmNvZGUvZ28vY2hhaW5jb2RlX2V4YW1wbGUwMhKAATdiZTE1MjllZTE2OTY5YmFmOWYzMTU2MjQ3YTBlZThl
N2VlZTk5YTZhMGE4MTY3NzZhY2ZmNjVlNmUxZGVmNzEyNDlmNGNiMWNhZDVlMGYwYjYwYjI1ZGQyYTY5NzVlZm
IyODI3NDFjMGUxZWNjNTNmYThjMTBhOWFhYTMxMTM3",
            "payload": "Cu0BCAESzAEKR2dpdGh1Yi5jb20vaHlwZXJsZWRnZXIvZmFicmljL2V4YW1wbG
VzL2NoYWluY29kZS9nby9jaGFpbmNvZGVfZXhhbXBsZTAyEoABN2JlMTUyOWVlMTY5NjliYWY5ZjMxNTYyNDdh
MGVlOGU3ZWVlOTlhNmEwYTgxNjc3NmFjZmY2NWU2ZTFkZWY3MTI0OWY0Y2IxY2FkNWUwZjBiNjBiMjVkZDJhNj
k3NWVmYjI4Mjc0MWMwZTFlY2M1M2ZhOGMxMGE5YWFhMzExMzcaGgoEaW5pdBIBYRIFMTAwMDASAWISBTIwMDAw"
,
            "timestamp": {
                "nanos": 298275779,
                "seconds": 1466057529
            },
            "type": 1,
            "uuid": "7be1529ee16969baf9f3156247a0ee8e7eee99a6a0a816776acff65e6e1def712
49f4cb1cad5e0f0b60b25dd2a6975efb282741c0e1ecc53fa8c10a9aaa31137"
        }
    ]
}
```


世界观

世界观用于存放链码执行过程中涉及到的状态变量，是一个键值数据库。典型的元素为 `[chaincodeID, ckey]: value` 结构。

为了方便计算变更后的 hash 值，一般采用默克尔树数据结构进行存储。树的结构由两个参数（ `numBuckets` 和 `maxGroupingAtEachLevel` ）来进行初始配置，并由 `hashFunction` 配置决定存放键值到叶子节点的方式。显然，各个节点必须保持相同的配置，并且启动后一般不建议变动。

- `numBuckets` ：叶子节点的个数，每个叶子节点是一个桶（bucket），所有的键值被

`hashFunction` 散列分散到各个桶，决定树的宽度；
- `maxGroupingAtEachLevel`：决定每个节点由多少个子节点的 hash 值构成，决定树的深度。

其中，桶的内容由它所保存到键值先按照 chaincodeID 聚合，再按照升序方式组成。

一般地，假设某桶中包括 个 chaincodeID，对于 ，假设其包括 个键值对，则聚合 G_i 内容可以计算为：

该桶的内容则为

注：这里的 `+` 代表字符串拼接，并非数学运算。

链码服务

链码包含所有的处理逻辑，并对外提供接口，外部通过调用链码接口来改变世界观。

接口和操作

链码需要实现 Chaincode 接口，以被 VP 节点调用。

```
type Chaincode interface { Init(stub *ChaincodeStub, function string, args []string) ([
]byte, error) Invoke(stub *ChaincodeStub, function string, args []string) ([]byte, err
or) Query(stub *ChaincodeStub, function string, args []string) ([]byte, error)}
```

链码目前支持的交易类型包括：部署（Deploy）、调用（Invoke）和查询（Query）。

- 部署：VP 节点利用链码创建沙盒，沙盒启动后，处理 protobuf 协议的 shim 层一次性发送包含 ChaincodeID 信息的 REGISTER 消息给 VP 节点，进行注册，注册完成后，VP 节点通过 gRPC 传递参数并调用链码 Init 函数完成初始化；
- 调用：VP 节点发送 TRANSACTION 消息给链码沙盒的 shim 层，shim 层用传过来的参数调用链码的 Invoke 函数完成调用；
- 查询：VP 节点发送 QUERY 消息给链码沙盒的 shim 层，shim 层用传过来的参数调用链码的 Query 函数完成查询。

不同链码之间可能互相调用和查询。

容器

在实现上，链码需要运行在隔离的容器中，超级账本采用了 Docker 作为默认容器。

对容器的操作支持三种方法：build、start、stop，对应的接口为 VM。

```
type VM interface {
  build(ctxt context.Context, id string, args []string, env []string, attachstdin bool,
 attachstdout bool, reader io.Reader) error
  start(ctxt context.Context, id string, args []string, env []string, attachstdin bool,
 attachstdout bool) error
  stop(ctxt context.Context, id string, timeout uint, dontkill bool, dontremove bool)
error
}
```

链码部署成功后，会创建连接到部署它的 VP 节点的 gRPC 通道，以接受后续 Invoke 或 Query 指令。

gRPC 消息

VP 节点和容器之间通过 gRPC 消息来交互。消息基本结构为

```
message ChaincodeMessage {

 enum Type { UNDEFINED = 0; REGISTER = 1; REGISTERED = 2; INIT = 3; READY = 4; TRANSAC
TION = 5; COMPLETED = 6; ERROR = 7; GET_STATE = 8; PUT_STATE = 9; DEL_STATE = 10; INVO
KE_CHAINCODE = 11; INVOKE_QUERY = 12; RESPONSE = 13; QUERY = 14; QUERY_COMPLETED = 15;
 QUERY_ERROR = 16; RANGE_QUERY_STATE = 17; }

 Type type = 1; google.protobuf.Timestamp timestamp = 2; bytes payload = 3; string uui
d = 4;}
```

当发生链码部署时，容器启动后发送 `REGISTER` 消息到 VP 节点。如果成功，VP 节点返回 `REGISTERED` 消息，并发送 `INIT` 消息到容器，调用链码中的 Init 方法。

当发生链码调用时，VP 节点发送 `TRANSACTION` 消息到容器，调用其 Invoke 方法。如果成功，容器会返回 `RESPONSE` 消息。

类似的，当发生链码查询时，VP 节点发送 `QUERY` 消息到容器，调用其 Query 方法。如果成功，容器会返回 `RESPONSE` 消息。

成员权限管理

通过基于 PKI 的成员权限管理，平台可以对接入的节点和客户端的能力进行限制。

证书有三种，Enrollment，Transaction，以及确保安全通信的 TLS 证书。

- 注册证书 ECert：颁发给提供了注册凭证的用户或节点，一般长期有效；
- 交易证书 TCert：颁发给用户，控制每个交易的权限，一般针对某个交易，短期有效。
- 通信证书 TLSCert：控制对网络的访问，并且防止窃听。



新的架构设计

目前，VP 节点执行了所有的操作，包括接收交易，进行交易验证，进行一致性达成，进行账本维护等。这些功能的耦合导致节点性能很难进行扩展。

新的思路就是对这些功能进行解耦，让每个功能都相对单一，容易进行扩展。社区内已经有了一些讨论。

Fabric 1.0 的设计采用了适当的解耦，根据功能将节点角色解耦开，让不同节点处理不同类型的工作负载。

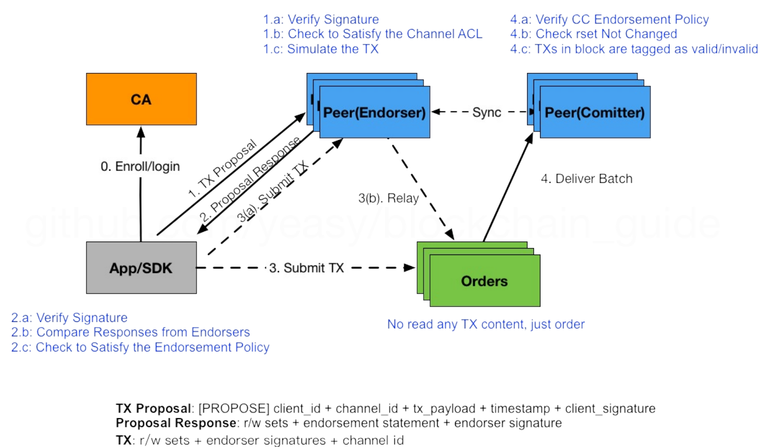


图 *1.14.2.1* - 示例工作过程

- 客户端：客户端应用使用 SDK 来跟 Fabric 打交道，构造合法的交易提案提交给 endorser；收集到足够多 endorser 支持后可以构造合法的交易请求，发给 orderer 或代理节点。
- Endorser peer：负责对来自客户端的交易进行合法性和 ACL 权限检查（模拟交易），通过则签名并返回结果给客户端。
- Committer peer：负责维护账本，将达成一致顺序的批量交易结果进行状态检查，生成区块，执行合法的交易，并写入账本，同一个物理节点可以同时担任 endorser 和 committer 的 角色。
- Orderer：仅负责排序，给交易们一个全局的排序，一般不需要跟账本和交易内容打交道。
- CA：负责所有证书的维护，遵循 PKI。

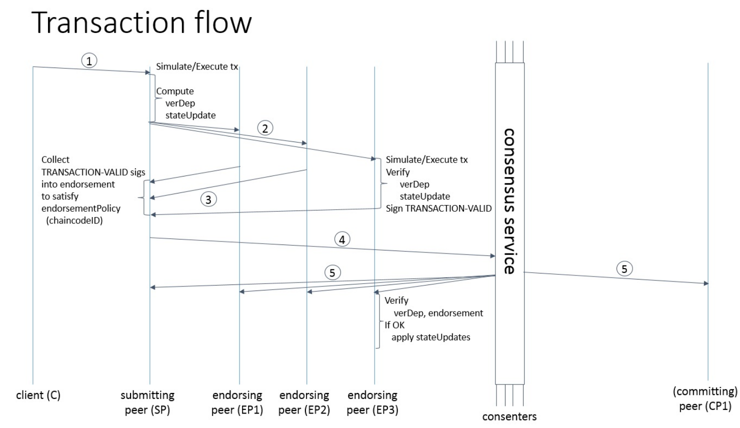


图 *1.14.2.2* - 示例交易过程

消息协议

节点之间通过消息来进行交互，所有消息都由下面的数据结构来实现。

```
message Message {
    enum Type {
        UNDEFINED = 0;

        DISC_HELLO = 1;
        DISC_DISCONNECT = 2;
        DISC_GET_PEERS = 3;
        DISC_PEERS = 4;
        DISC_NEWMSG = 5;

        CHAIN_STATUS = 6;
        CHAIN_TRANSACTION = 7;
        CHAIN_GET_TRANSACTIONS = 8;
        CHAIN_QUERY = 9;

        SYNC_GET_BLOCKS = 11;
        SYNC_BLOCKS = 12;
        SYNC_BLOCK_ADDED = 13;

        SYNC_STATE_GET_SNAPSHOT = 14;
        SYNC_STATE_SNAPSHOT = 15;
        SYNC_STATE_GET_DELTAS = 16;
        SYNC_STATE_DELTAS = 17;

        RESPONSE = 20;
        CONSENSUS = 21;
    }
    Type type = 1;
    bytes payload = 2;
    google.protobuf.Timestamp timestamp = 3;
}
```

消息分为四大类：Discovery（探测）、Transaction（交易）、Synchronization（同步）、Consensus（一致性）。

不同消息类型，对应到 payload 中数据不同，分为对应的子类消息结构。

Discovery

包括 DISC_HELLO、DISC_GET_PEERS、DISC_PEERS。

DISC_HELLO 消息结构如下。

```
message HelloMessage { PeerEndpoint peerEndpoint = 1; uint64 blockNumber = 2;}message
PeerEndpoint { PeerID ID = 1; string address = 2; enum Type { UNDEFINED = 0; VALIDATOR
= 1; NON_VALIDATOR = 2; } Type type = 3; bytes pkiID = 4;}

message PeerID { string name = 1;}
```

节点新加入网络时，会向 `CORE_PEER_DISCOVERY_ROOTNODE` 发送 `DISC_HELLO` 消息，汇报本节点的信息（id、地址、block 数、类型等），开始探测过程。

探测后发现 block 数落后对方，则会触发同步过程。

之后，定期发送 `DISC_GET_PEERS` 消息，获取新加入的节点信息。收到 `DISC_GET_PEERS` 消息的节点会通过 `DISC_PEERS` 消息返回自己知道的节点列表。

Transaction

包括 Deploy、Invoke、Query。消息结构如下：

```
message Transaction { enum Type { UNDEFINED = 0; CHAINCODE_DEPLOY = 1; CHAINCODE_INVOK
E = 2; CHAINCODE_QUERY = 3; CHAINCODE_TERMINATE = 4; } Type type = 1; string uuid = 5;
bytes chaincodeID = 2; bytes payloadHash = 3;

 ConfidentialityLevel confidentialityLevel = 7; bytes nonce = 8; bytes cert = 9; bytes
 signature = 10;

 bytes metadata = 4; google.protobuf.Timestamp timestamp = 6;}

message TransactionPayload { bytes payload = 1;}

enum ConfidentialityLevel { PUBLIC = 0; CONFIDENTIAL = 1;}
```

Synchronization

当节点发现自己 block 落后网络中最新状态，则可以通过发送如下消息（由 consensus 策略决定）来获取对应的返回。

- SYNC_GET_BLOCKS（对应 SYNC_BLOCKS）：获取给定范围内的 block 数据；
- SYNC_STATE_GET_SNAPSHOT（对应 SYNC_STATE_SNAPSHOT）：获取最新的世界观快照；
- SYNC_STATE_GET_DELTAS（对应 SYNC_STATE_DELTAS）：获取某个给定范围内的 block 对应的状态变更。

Consensus

consensus 组件收到 `CHAIN_TRANSACTION` 类消息后，将其转换为 `CONENSUS` 消息，然后向所有的 VP 节点广播。

收到 `CONSENSUS` 消息的节点会按照预定的 consensus 算法进行处理。

本章小结

区块链服务平台设计

规模是困难之源。

信息产业过去的十年，是云计算的十年。云计算技术为传统信息行业带来了前所未有的便捷。用户无需在意底层实现细节，通过简单的操作，即可获得可用的计算资源，节约大量运维管理的时间成本。

区块链平台作为分布式基础设施，其部署和维护过程需要多方面的技能，这对很多应用开发者来说都是不小的挑战。为了解决这些问题，区块链即服务（Blockchain as a Service, BaaS）平台应运而生。BaaS 可以利用云服务基础设施的部署和管理优势，为开发者提供创建、使用，甚至安全监控区块链平台的快捷服务。目前，业界已有一些区块链前沿技术团队率先开发并上线了区块链服务平台。

本章将首先介绍 BaaS 的概念，之后分别介绍业界领先的 IBM Bluemix 和微软 Azure 云上所提供的区块链服务。最后，还介绍了超级账本的区块链管理平台 —— Cello 项目，以及如何使用它快速搭建一套可以个性化的区块链服务平台。

简介

区块链即服务（Blockchain as a Service，BaaS），是部署在云计算基础设施之上，对外提供区块链网络的生命周期管理和运行时服务管理等功能的一套工具。

构建一套分布式的区块链方案绝非易事，既需要硬件基础设施的投入，也需要全方位的开发和运营管理（DevOps）。BaaS 作为一套工具，可以帮助开发者快速生成必要的区块链环境，进而验证所开发的上层应用。

除了区块链平台本身，一套完整的解决方案实际上还可以包括设备接入、访问控制、服务监控等管理功能。这些功能，让 BaaS 平台可以为开发者提供更强大的服务支持。

从 2016 年起，业界已有一些前沿技术团队发布了 BaaS 平台，除了商业的方案如 IBM Bluemix 和微软 Azure 云之外，超级账本开源项目也发起了 Cello 项目，以提供一套实现区块链平台运营管理功能的开源框架。

参考架构

下图给出了区块链即服务功能的参考架构，自上而下分为多层结构。最上层面向应用开发者和平台管理员提供不同的操作能力；核心层负责完成包括资源编排、系统监控、数据分析和权限管理等重要功能；下层可以通过多种类型的驱动和代理组件来访问和管理多种物理资源。

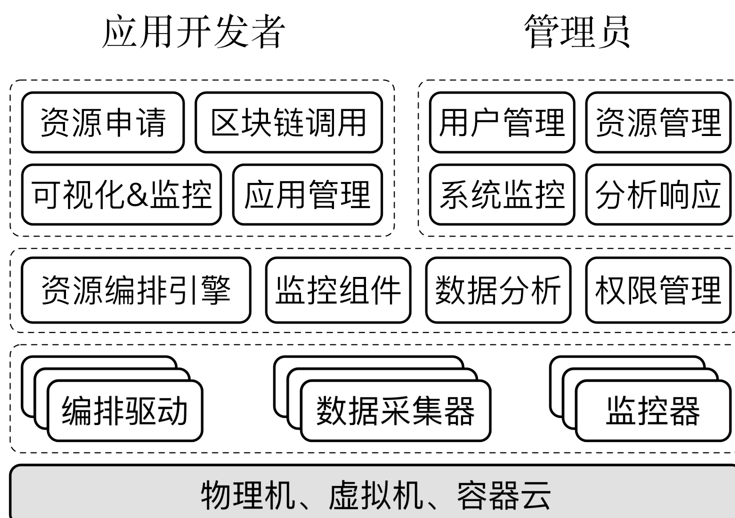


图 *1.15.1.1* - 区块链服务参考架构

典型地，BaaS 平台所提供的业务能力通常包括：

- 用户按需申请区块链网络，以及所需的计算、存储与网络连接资源；
- 用户对申请到的区块链进行生命周期管理，甚至支持灵活、弹性的区块链配置；
- 通过提供接口，让用户自由访问所申请到的区块链网络并进行调用；
- 提供直观的区块链可视化监控与操作界面，将区块链应用与底层平台无缝对接；
- 提供简单易用的智能合约开发与测试环境，方便用户对应用代码进行管理；
- 为管理员提供用户管理和资源管理操作；
- 为管理员提供对系统各项健康状态的实时监控；
- 提供对平台内各项资源和应用层的数据分析和响应能力。

考量指标

对于 BaaS 服务提供方，搭建这样一套功能完善、性能稳定的 BaaS 平台存在诸多挑战。可以从如下几个角度进行考量设计。

- 性能保障：包括区块链和应用的响应速度，监控实时性等；
- 可扩展性：支持大规模场景下部署和管理的能力，可以快速进行扩展；
- 资源调度：对于非均匀的资源请求类型可以智能的予以平缓化处理，合理分配系统资源；
- 安全性：注意平衡用户操作区块链的自由度与平台自身的安全可控；
- 可感知性：深度感知数据行为，如可以准确实时评估区块链的运行状况，给用户启发；
- 底层资源普适性：底层应当支持多种混合计算架构，容易导入物理资源。

此外，对于面向开发者的 BaaS 服务，创建的区块链环境应当尽量贴近实际应用场景，让用户可以将经过检验的区块链模型很容易地迁移到生产环境。甚至可以直接联动支持第三方发布平台，直接将经过验证的应用推向发布环境。

IBM Bluemix 云区块链服务

Bluemix 是 IBM 推出的开放的 PaaS 云平台，包含大量平台和软件服务，旨在帮助开发者实现一站式地应用开发与部署管理。

2016 年，Bluemix 面向开发者推出了基于超级账本 Fabric 的区块链服务，供全球的区块链爱好者使用。用户可以通过访问 https://console.ng.bluemix.net/catalog/services/blockchain 使用该服务。

服务介绍

Bluemix 为用户提供了在云上灵活管理超级账本 Fabric 区块链网络的能力，让开发者专注于快速创建、操作和监控区块链网络，而无需过多考虑底层硬件资源。同时，Bluemix 云平台本身也提供了安全、隐私性方面的保障，并对相关资源进行了性能优化。

Bluemix 目前提供了几种不同类型的区块链网络部署方案，包括免费的基础套餐到收费的高性能方案等。不同方案针对开发者的不同需求，在运行环境、占用资源、配置方式上都有所区别。

对于超级账本 Fabric 网络试用者，可选择免费的基础套餐，获得一个包含各类型 Peer 节点和 CA 的完整区块链试用网络，用户可以自行尝试部署链码并实时观察账本状态的变化。

使用服务

Bluemix 云平台提供的仪表盘（Dashboard）提供了十分直观的管理方式，用户可以通过 Web 界面来获取和访问区块链资源。

如下图所示，用户创建网络后，可以进入 Dashboard 看到属于自己的区块链网络，同时观察各节点的状态，以及与身份认证相关的服务凭证。

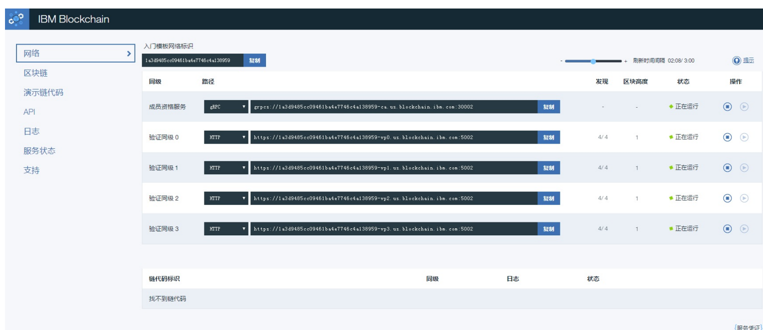


图 *1.15.2.1 - Bluemix* 区块链服务仪表盘

对于已经申请到的区块链网络，用户可以通过 Dashboard 对其部署并调用链码，并实时查看响应结果。例如，下图中展示了部署自带的 example02 链码。



图 *1.15.2.2* - 通过 *Dashboard* 操作链码

对链码的操作会发送交易，进而生成新的区块。可通过 Dashboard 观察与区块链状态、区块内容相关的信息。例如，下图中区块链生成了 4 个区块，并执行了 1 次部署和 2 次调用。

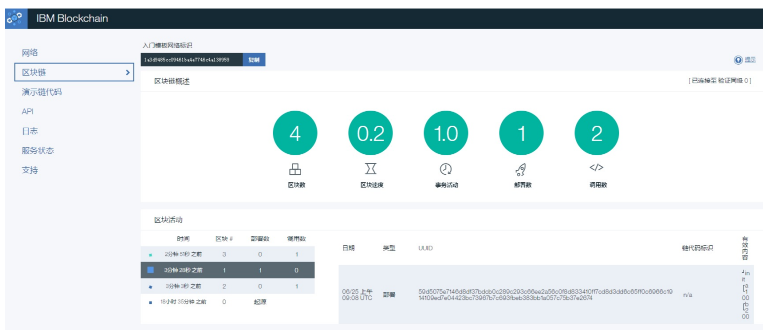


图 *1.15.2.3* - 通过 *Dashboard* 观察区块链

平台同时会收集各节点的日志信息，监控和记录服务的运行状态。用户同样可以在 Dashboard 中实时查看。如下图所示，显示了服务和网络的正常运行时间等。

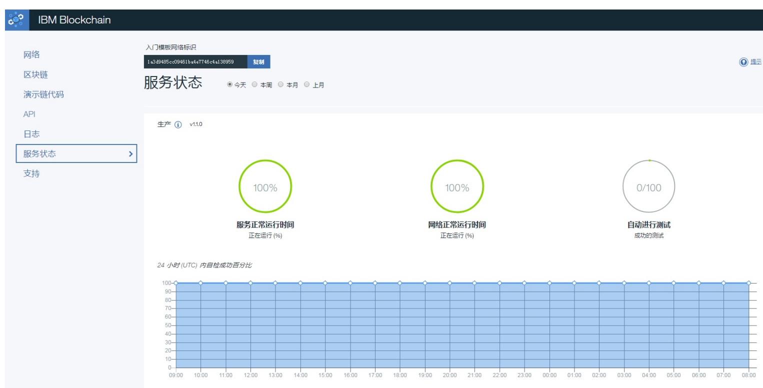


图 *1.15.2.4* - 通过 *Dashboard* 获取服务状态

同时，Bluemix 云平台会将与区块链网络交互所需的 gRPC 或 HTTP 接口地址开放给用户，供用户通过 SDK 等进行远程操作，实现更多跟区块链、链码和应用相关的丰富功能。

微软 Azure 云区块链服务

Azure 是微软推出的云计算平台，向用户提供开放的 IaaS 和 PaaS 服务。

Azure 陆续在其应用市场中提供了若干个与区块链相关的服务，分别面向多种不同的区块链底层平台，其中包括以太坊和超级账本 Fabric。

可以在应用市场（https://azuremarketplace.microsoft.com/en-us/marketplace/apps）中搜索 “blockchain” 关键字查看这些服务，如下图所示。

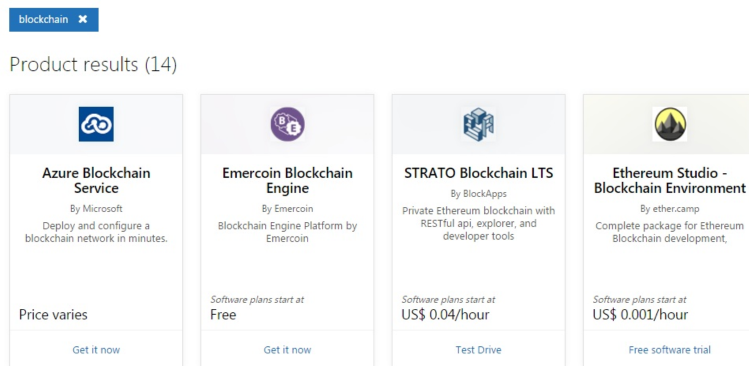


图 *1.15.3.1 - Azure* 上的区块链服务

下面具体介绍其中的 Azure Blockchain Service。

使用服务

使用 Azure 服务，用户可以在几分钟之内在云中部署一个区块链网络。云平台会将一些耗时的配置流程自动化，使用户专注在上层应用方案。

Azure 区块链服务目前支持部署以太坊或超级账本 Fabric 网络。

下面以以太坊为例，在 Azure 的仪表盘中，选择创建 Ethereum Consortium Blockchain 后，输入一些配置选项，则可以开始部署该模拟网络，如下图所示。

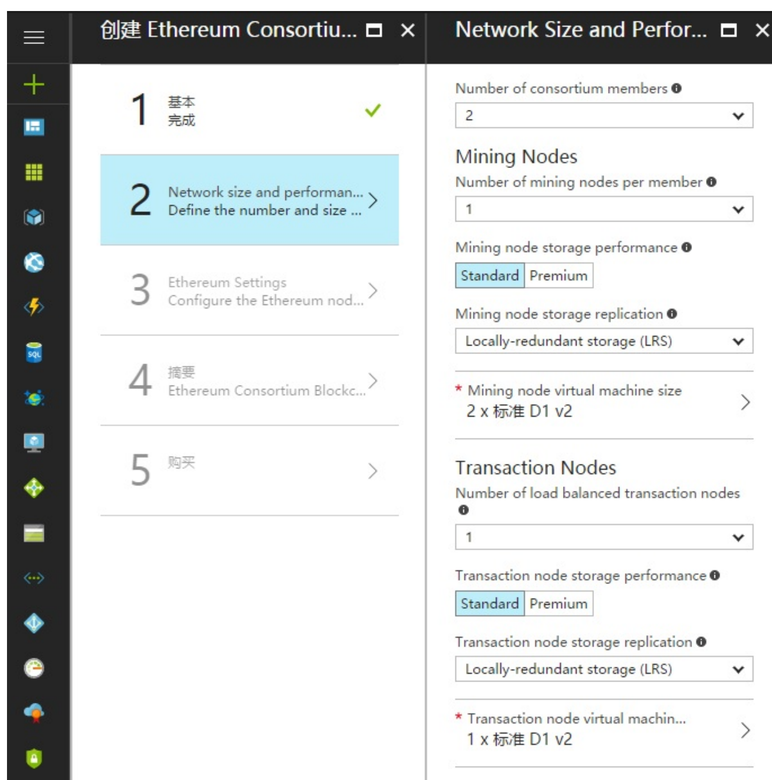


图 *1.15.3.2 - Azure* 区块链配置

部署过程需要几分钟时间。完成后，可进入资源组查看部署结果，如下图所示，成功部署了一个以太坊网络。

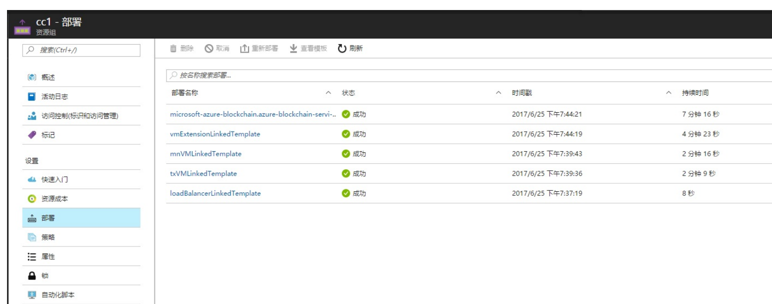


图 *1.15.3.3 - Azure* 区块链部署结果

点击 microsoft-azure-blockchain 开头的链接，可以查看网络的一些关键接口，包括管理网址、RPC 接口地址等。

复制管理网址 ADMIN-SITE 的链接，用浏览器打开，可以进入区块链管理界面。界面中可查看网络各节点信息，也可以新建一个账户，并从 admin 账户向其发送 1000 个以太币。结果如下图所示。

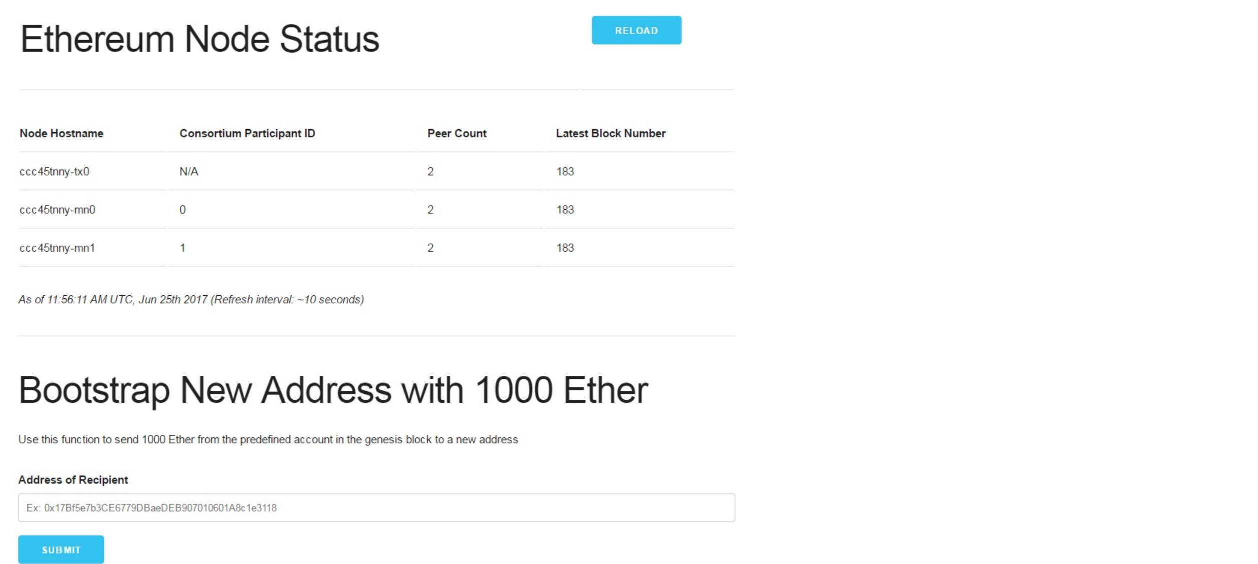


图 *1.15.3.4 - Azure* 区块链管理界面

Azure 云平台提供了相对简单的操作界面，更多的是希望用户通过 RPC 接口地址来访问所部署的区块链示例。用户可以自行通过 RPC 接口与以太坊模拟网络交互，部署和测试智能合约，此处不再赘述。

使用超级账本 Cello 搭建区块链服务

从前面的讲解中可以看到，区块链服务平台能够有效加速对区块链技术的应用，解决企业和开发者进行手动运营管理的负担。但是这些方案都是商业用途，并且只能在线使用。

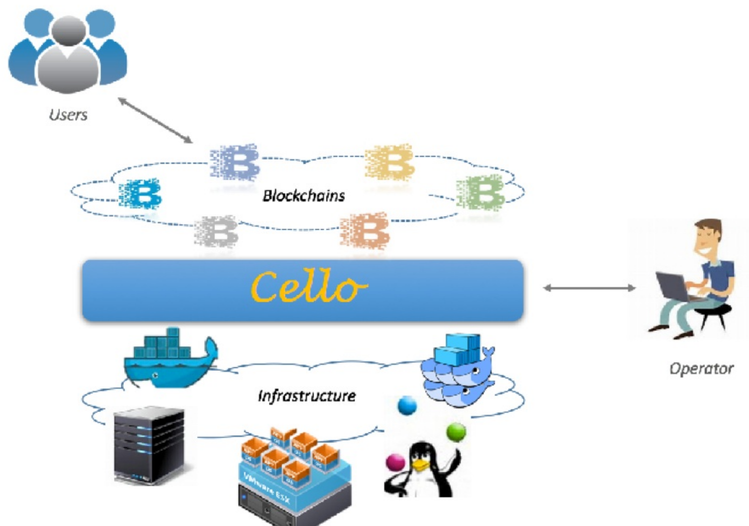


图 *1.15.4.1 - Cello* 典型应用场景

超级账本的 Cello 项目为本地搭建区块链服务管理平台提供了开源的解决方案，可以实现在多种类型的物理资源上实现区块链网络的生命周期管理。

正如 Cello 的名字所蕴意，它就像一把精巧的大提琴，以区块链为琴弦，可以奏出更加动人的乐章。

基本架构和特性

Cello 项目由笔者领导的 IBM 技术团队于 2017 年 1 月贡献到超级账本社区，主要基于 Python 和 Javascript 语言编写。该项目的定位为区块链管理平台，支持部署、运行时管理和数据分析等功能，可以实现一套完整的 BaaS 系统的快速搭建。其基本架构如下图所示。

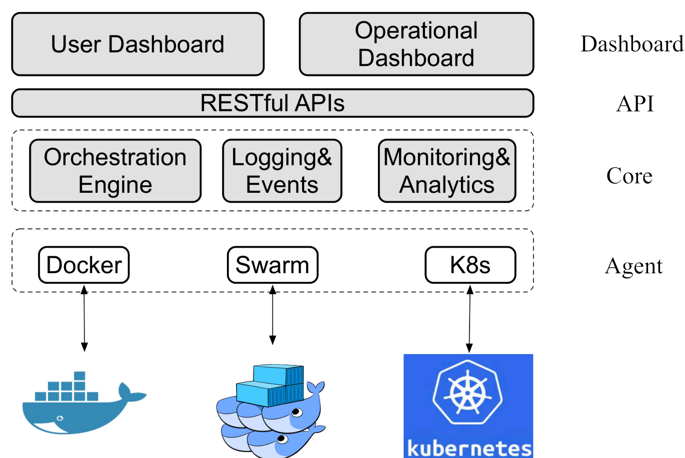


图 *1.15.4.2 - Cello* 基本架构

在实现区块链环境快速部署的同时，Cello 也提供了不少对区块链平台进行运行时管理的特性，这些特性总结如下。

- 管理区块链的全生命周期，包括创建、配置、使用、健康检查、删除等。
- 支持多种基础架构作为底层资源池，包括裸机、虚拟机、容器云（Docker、Swarm、Kubernetes）等。
- 支持多种区块链平台及自定义配置（目前以支持超级账本 Fabric 为主）。
- 支持监控和分析功能，实现对区块链网络和智能合约的运行状况分析。
- 提供可插拔的框架设计，包括区块链平台、资源调度、监控、驱动代理等都很容易引入第三方实现。

下面具体介绍如何以 Docker 主机为资源池，用 Cello 快速搭建一个区块链服务平台。

环境准备

Cello 采用了典型的主从（Master-Worker）架构。用户可以自行准备一个 Master 物理节点和若干个 Worker 节点。

其中，Master 节点负责管理（例如，创建和删除）Worker 节点中的区块链集群，其通过 8080 端口对外提供网页 Dashboard，通过 80 端口对外提供 RESTful API。Worker 节点负责提供区块链集群的物理资源，例如基于 Docker 主机或 Swarm 的方式启动多个集群，作为提供给用户可选的多个区块链网络环境。

下图中展示了一个典型的 Master-Worker 部署拓扑。每个节点默认为 Linux（如 Ubuntu 16.04）服务器或虚拟机。

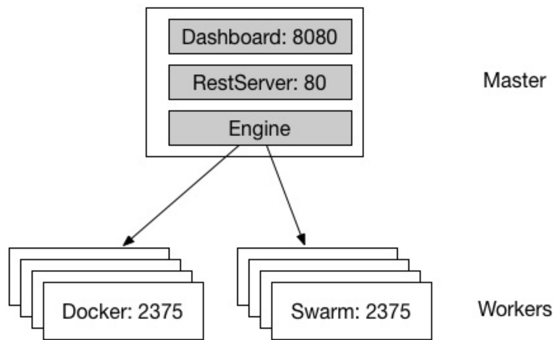


图 *1.15.4.3 - Cello* 部署拓扑示例

为了支持区块链网络，Worker 和 Master 节点需要配备足够的物理资源。例如，如果希望在一个 Worker 节点上能够启动至少 10 个区块链集群，则建议节点配置至少为 8 CPU、16G 内存、100G 硬盘容量。

下载 Cello 源码

Cello 代码的官方仓库在社区的 gerrit 上，并实时同步到 Github 仓库中，读者可以从任一仓库中获取代码。例如通过如下命令从官方仓库下载 Cello 源码。

```
$ git clone http://gerrit.hyperledger.org/r/cello && cd cello
```

配置 Worker 节点

安装和配置 Docker 服务

首先安装 Docker，推荐使用 1.12 或者更新的版本。可通过如下命令快速安装 Docker。

```
$ curl -fsSL https://get.docker.com/ | sh
```

安装成功后，修改 Docker 服务配置。对于 Ubuntu 16.04，更新 `/lib/systemd/system/docker.service` 文件如下。

```
[Service]
DOCKER_OPTS="$DOCKER_OPTS -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --api-c
ors-header='*' --default-ulimit=nofile=8192:16384 --default-ulimit=nproc=8192:16384"
EnvironmentFile=-/etc/default/docker
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// $DOCKER_OPTS
```

修改后，需要通过如下命令重启 Docker 服务。

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker.service
```

下载 Docker 镜像

对于超级账本 Fabric v1.0 集群所需的镜像，可以使用如下命令进行自动下载。

```
$ cd scripts/worker_node_setup && bash download_images.sh
```

防火墙配置

为了确保 Worker 上的容器可以正常访问，通过如下命令确保主机开启 IP 转发。

```
$ sysctl -w net.ipv4.ip_forward=1
```

同时检查主机的 iptables 设置，确保必要的端口被打开（如 2375、7050~10000 等）。

配置 Master 节点

下载 Docker 镜像

使用如下命令下载运行服务所必要的 Docker 镜像。

其中，python:3.5 镜像是运行 Cello 核心组件的基础镜像；mongo:3.2 提供了数据库服务；yeasy/nginx:latest 提供了 Nginx 转发功能；mongo-express:0.30 镜像是为了调试数据库，可以选择性安装。

```
$ docker pull python:3.5 \
    && docker pull mongo:3.2 \
    && docker pull yeasy/nginx:latest \
    && docker pull mongo-express:0.30
```

安装 Cello 服务

首次运行时，可以通过如下命令对 Master 节点进行快速配置，包括安装 Docker 环境、创建本地数据库目录、安装依赖软件包等。

```
$ make setup
```

如果安装过程没有提示出现问题，则说明当前环境满足了运行条件。如果出现问题，可通过查看日志信息进行定位。

管理 Cello 服务

可以通过运行如下命令来快速启动 Cello 相关的组件服务（包括 dashboard、restserver、watchdog、mongo、nginx 等）。

```
$ make start
```

类似地，运行 `make stop` 或 `make restart` 可以停止或重启全部服务。

若希望重新部署某个特定服务（如 dashboard），可运行如下命令。

```
$ make redeploy service=dashboard
```

运行如下命令可以实时查看所有服务的日志信息。

```
$ make logs
```

若希望查看某个特定服务的日志，可运行如下命令进行过滤，如只查看 watchdog 组件的日志。

```
$ make log service=watchdog
```

使用 Cello 管理区块链

Cello 服务启动后，管理员可以通过 Cello 的 Dashboard 页面管理区块链。

默认情况下，可通过 Master 节点的 8080 端口访问 Dashboard。默认的登录用户名和密码为 `admin:pass` 。

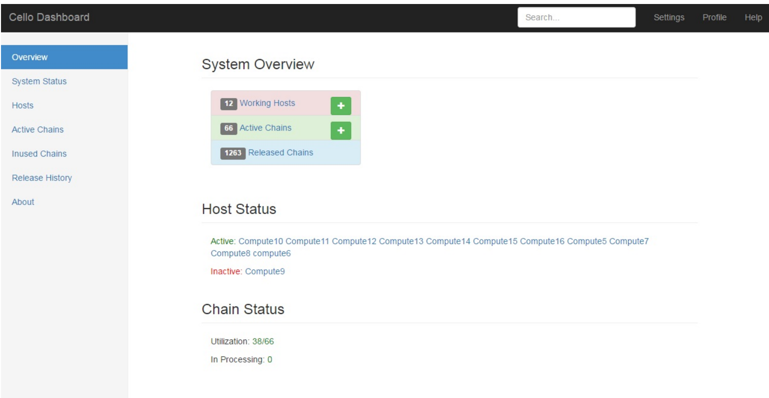


图 *1.15.4.4 - Cello Dashboard*

如图，Dashboard 有多个页面，各页面的功能如下。

页面	功能
Overview	展示系统整体状态
System Status	展示一些统计信息
Hosts	管理所有主机（Worker 节点）
Active Chains	管理资源池中的所有链
Inused Chains	管理正在被用户占用的链
Released History	查看链的释放历史

Hosts 页面

在 Hosts 页面，管理员可以管理所有资源池中已存在的主机，或添加新主机。表格中会显示主机的类型、状态、正在运行的区块链数量、区块链数量上限等。所有设定为 non-schedulable (不会自动分配给用户）的主机会用灰色背景标识，如下图所示。

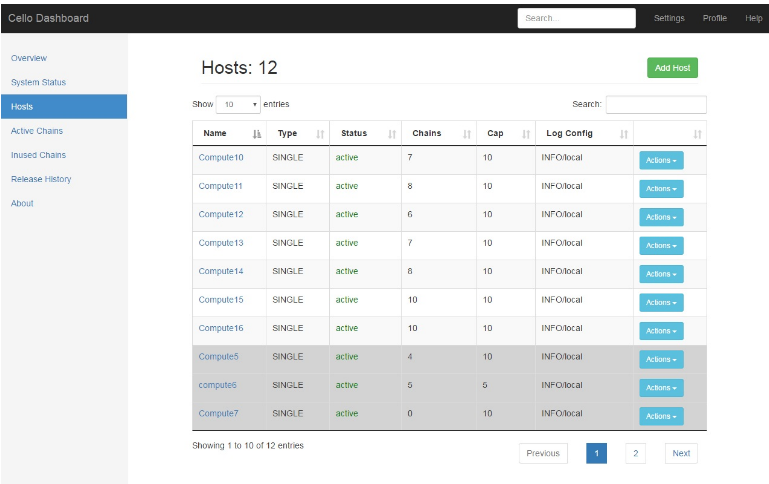


图 *1.15.4.5 - Hosts* 页面

点击一个主机的 Action 下拉菜单，有如下选项可供操作该主机。

- Fillup：将主机运行的区块链数添加至上限。
- Clean：清理主机中所有未被用户占用的链。
- Config：更改主机配置，如名称和链数量上限。
- Reset：重置该主机，只有当该主机没有用户占用的链时可以使用。
- Delete：从资源池中删除该主机。

点击 Hosts 页面的 Add Host 按钮，可以向资源池中添加主机。需要设定该主机的名称、Daemon URL 地址（例如，Worker 节点的 docker daemon 监听地址和端口）、链数量上限、日志配置、是否启动区块链至数量上限、是否可向用户自动分配，如下图所示。

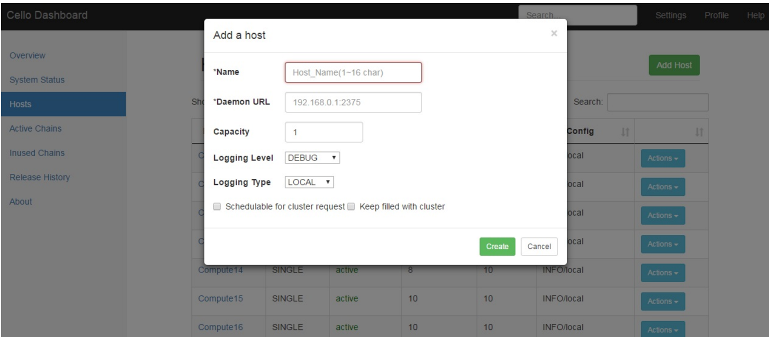


图 *1.15.4.6 -* 添加主机

Active Chains 页面

Active Chains 页面会显示所有正在运行的链，包括链的名称、类型、状态、健康状况、规模、所属主机等信息。正在被用户占用的链会用灰色背景标识，如下图所示。

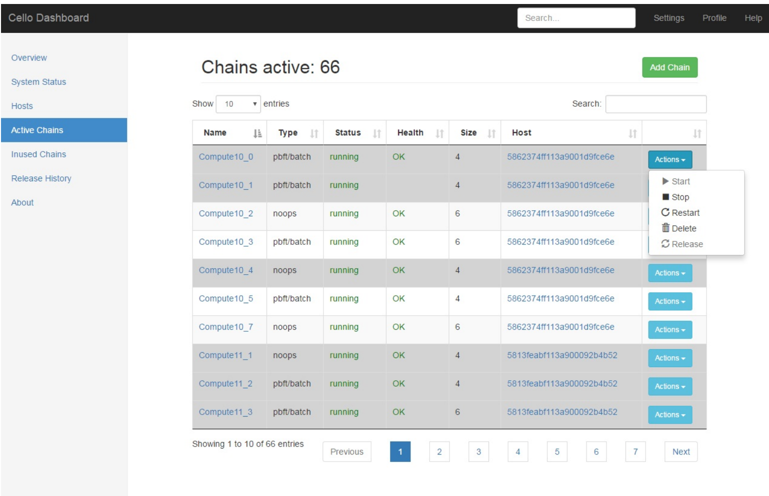


图 *1.15.4.7 - Active Chains* 页面

点击一条链的 Actions 下拉菜单，有如下选项可供操作该链。

- Start：如果这条链处于停止状态，则启动。
- Stop：停止运行中的链。
- Restart：重新启动这条链。
- Delete：删除这条链。
- Release：将占用的链释放，随后会被删除。

点击 Active Chains 页面的 Add Chain 按钮，可以向资源池中添加更多链（如果还有未被占满的主机），如下图所示。

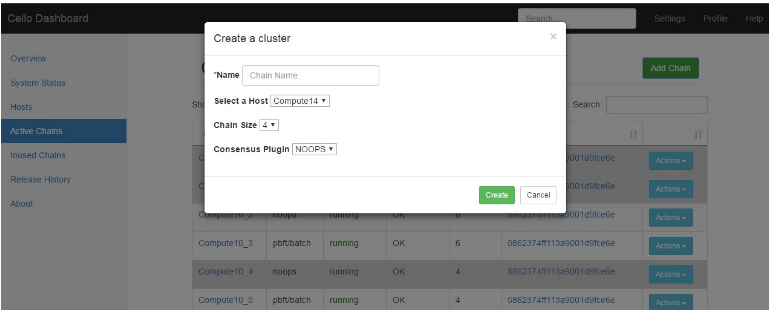


图 *1.15.4.8* - 添加链

基于 **Cello** 进行功能扩展

Cello 已经提供了完整的区块链管理功能，并提供了图形界面和 API。

用户可以通过向 Cello 的 Master 节点（默认为 80 端口）发送 RESTful API 来申请、释放区块链，或查看区块链相关信息，如其对外开放的接口，可供用户进行远程交互。RESTful API 的说明可在 Cello 的文档中查阅。

对于区块链服务提供者，可以利用这些 API 为用户呈现友好的区块链申请和操作界面，在 Cello 的基础之上构建和实现更多功能。

本章小结

本章介绍了区块链即服务的概念，阐述了整合云计算技术能够为区块链部署和管理所带来的便捷。接下来提出了区块链服务平台的参考架构，并从功能和性能等实践角度总结了平台设计的考量指标。

本章随后还介绍了业界领先的 IBM Bluemix 和微软 Azure 云上提供的区块链服务。最后讲解了如何使用超级账本 Cello 项目快速搭建一套个性化的区块链服务平台。

区块链技术的普及离不开生态系统和相关工具的成熟，区块链应用的落地同样离不开完善的 DevOps 支持。本章的内容能够给予读者不同的视角，从系统方案的角度出发，思考如何在新技术变革中保持应对变化的敏捷与高效。

性能与评测

过早优化，往往引来各种麻烦。

一项技术究竟能否实用，有两项基本指标十分关键：一是功能的完备；一是性能的达标。

本章将试图对已有区块链技术进行一些评测。所有结果将尽可能保证客观准确，但不保证评测方法是否科学、评测结果是否具备足够参考性。

简介

区块链的平台性能跟很多因素都有关系，特别在实际应用中，根据应用场景的不同和系统设计和使用的不同，可能同一套平台最终在业务体现上会有较大差异。

在这里，仅侧重评测一般意义上的平台性能。

所有给出指标和结果仅供参考，由于评测环境和方案不同，不保证结果的一致性。

生产环境中应用区块链技术请务必进行充分验证评测。

Hyperledger Fabric v0.6 性能评测

环境配置

类型	操作系统	内核版本	CPU(GHz)	内存(GB)
物理机	Ubuntu 14.04.1	3.16.0-71-generic	4x2.0	8

每个集群启动后等待 10s 以上，待状态稳定。

仅测试单客户端、单服务端的连接性能情况。

评测指标

一般评测系统性能指标包括吞吐量（throughput）和延迟（latency）。对于区块链平台系统来说，实际交易延迟包括客户端到系统延迟（往往经过互联网），再加上系统处理反馈延迟（跟不同 consensus 算法关系很大，跟集群之间互联系统关系也很大）。

本次测试仅给出大家最为关注的交易吞吐量（tps）。

结果

query 交易

noops

clients	VP Nodes	iteration	tps
1	1	2000	195.50
1	4	2000	187.09

pbft:classic

clients	VP Nodes	iteration	tps
1	4	2000	193.05

pbft:batch

clients	VP Nodes	batch size	iteration	tps
1	4	2	2000	193.99
1	4	4	2000	192.49
1	4	8	2000	192.68

pbft:sieve

clients	VP Nodes	iteration	tps
1	4	2000	192.86

invoke 交易

noops

clients	VP Nodes	iteration	tps
1	1	2000	298.51
1	4	2000	205.76

pbft:classic

clients	VP Nodes	iteration	tps
1	4	2000	141.34

pbft:batch

clients	VP Nodes	batch size	iteration	tps
1	4	2	2000	214.36
1	4	4	2000	227.53
1	4	8	2000	237.81

pbft:sieve

clients	VP Nodes	iteration	tps
1	4	2000	253.49*

注：*sieve* 算法目前在所有交易完成后较长时间内并没有取得最终的结果，出现大量类似*“vp0_1 | 07:49:26.388 [consensus/obcpbft] main -> WARN 23348 Sieve replica 0 custody expired, complaining:*

*3kwyMkdCSL4rbajn65v+iYWyJ5aqagXvRR9QU8qezpAZXY4y6uy2MB31SGaAiaSyPMM77 TYADdBmAaZveM38zA==”*警告信息。

结论

单客户端连接情况下，tps 基本在 190 ~ 300 范围内。

小结

附录

术语

通用术语

- Blockchain（区块链）：基于密码学的可实现信任化的信息存储和处理的结构和技术。
- Byzantine Failure（拜占庭错误）：指系统中存在除了消息延迟或不可送达的故障以外的错误，包括消息被篡改、节点不按照协议进行处理等，潜在地会对系统造成针对性的破坏。
- CDN：Content Delivery Network，内容分发网络。利用在多个地理位置预先配置的缓存服务器，自动从距离近的缓存服务器进行对请求的响应，以实现资源的快速分发。
- Consensus（共识）：分布式系统中多个参与方对某个信息达成一致，多数情况下为对发生事件的顺序达成一致。
- Decentralization（去中心化）：无需一个独立第三方的中心机构存在，有时候也叫多中心化。
- Distributed（分布式）：非单体中央节点的实现，通常由多个个体通过某种组织形式联合在一起，对外呈现统一的服务形式。
- Distributed Ledger（分布式账本）：由多家联合维护的去中心化（或多中心化）的账本记录平台。
- DLT：Distributed Ledger Technology，分布式账本技术。包括区块链、权限管理等在内的实现分布式账本的技术。
- DTCC：Depository Trust and Clearing Corporation，存托和结算公司。全球最大的金融交易后台服务机构。
- Fintech：Financial Technology，跟金融相关的（信息）技术。
- Gossip：一种 P2P 网络中多个节点之间进行数据同步的协议，如随机选择邻居进行转发。
- LDAP：Lightweight Directory Access Protocol，轻量级目录访问协议，是一种为查询、搜索业务而设计的分布式数据库协议，一般具有优秀的读性能，但写性能往往较差。
- Market Depth（市场深度）：衡量市场承受大额交易后汇率的稳定能力，例如证券交易市场出现大额交易后价格不出现大幅波动。
- MTBF：Mean Time Between Failures，平均故障间隔时间，即系统可以无故障运行的预期时间。
- MTTR：Mean Time to Repair，平均修复时间，即发生故障后，系统可以恢复到正常运行的预期时间。
- MVCC：Multi-Version Concurrency Control，多版本并发控制。数据库领域的技术，通过引入版本来实现并发更新请求的乐观处理，当更新处理时数据版本跟请求中注明版本不一致时则拒绝更新。发生更新成功则将数据的版本加一。
- Non-validating Peer（非验证节点）：不参与账本维护，仅作为交易代理响应客户端的请求，并对交易进行一些基本的有效性检查，之后转发给验证节点。

- P2P：点到点的通信网络，网络中所有节点地位均等，不存在中心化的控制机制。
- SLA/SLI/SLO：Service Level Agreement/Indicator/Objective，分别描述服务可用性对用户的承诺，功能指标和目标值。
- SWIFT：Society for Worldwide Interbank Financial Telecommunication，环球银行金融电信协会，运营世界金融电文网络，服务银行和金融机构。
- Turing-complete（图灵完备）：指一个机器或装置能用来模拟图灵机（现代通用计算机的雏形）的功能，图灵完备的机器在可计算性上等价。
- Validating Peer（验证节点）：维护账本的核心节点，参与一致性维护、对交易的验证和执行。更进一步可以划分为 Endorser、Committer 等多种角色。

密码学与安全相关

- ASN.1：Abstract Syntax Notation One，定义了描述数据的表示、编码、传输、解码的一套标准，被广泛应用在计算机、通信和安全领域。
- CA：Certificate Authority，负责证书的创建、颁发，在 PKI 体系中最为核心的角色。
- CRL：Certification Revocation List，证书吊销列表，包含所撤销的证书列表。
- CSR：Certificate Signing Request，证书签名申请，包括通用名、名称、主机、生成私钥算法和大小、CA 配置和序列号等信息，用来发给 CA 服务以颁发签名的证书。
- DER：Distinguished Encoding Rules，ASN.1 中定义的一种二进制编码格式，可以用来保存证书或密钥内容。
- Genesis Block：创世区块，区块链的第一个区块，一般用于初始化，不带有交易信息。
- Hash：哈希算法，任意长度的二进制值映射为较短的固定长度的二进制值的算法。
- IES：Integrated Encryption Scheme，集成加密机制，一种混合加密机制，可以应对选择明文攻击（可以获知任意明文和对应密文）情况下的攻击。包括 DLIES（基于离散对数）和 ECIES（基于椭圆曲线）两种实现。
- Nonce：密码学术语，表示一个临时的值，多为随机字符串。
- OCSP：Online Certificate Status Protocol，在线证书状态协议，通过查询服务来在线确认证书的状态（如是否撤销）。RFC 2560 中定义。
- PKCS：Public-Key Cryptography Standards，公钥密码标准，由 RSA 实验室提出，定义了利用 RSA 算法和相关密码学技术来实现安全的系列规范，目前包括 15 个不同领域的规范。最早的版本在 1991 年提出，目前最新版本为 2012 年提出的 2.2 版本。
- PEM：Privacy Enhanced Mail，用来保存证书和密钥的一种编码格式，RFC 1421-1424 中定义。
- PKI：Public Key Infrastructure，基于公钥体系的安全基础架构。
- SM：国家商用密码算法，2010 年以来陆续由国家密码管理局发布的相关标准和规范，主要包括：SM2（基于椭圆曲线密码的公钥密码算法标准）、SM3（Hash 算法标准）、SM4（基于分组加密的对称密码算法标准）、SM9（基于身份的数字证书体系）。

比特币、以太坊相关术语

- Bitcoin（比特币）：最早由中本聪提出和实现的基于区块链思想的数字货币技术。
- DAO：Decentralized Autonomous Organization，分布式自治组织，基于区块链的按照智能合约联系起来的松散自治群体。
- Lightning Network（闪电网络）：通过链外的微支付通道来增大交易吞吐量的技术。
- Mining（挖矿）：通过暴力尝试来找到一个字符串，使得它加上一组交易信息后的 Hash 值符合特定规则（例如前缀包括若干个 0），找到的人可以宣称新区块被发现，并获得系统奖励的数字货币。
- Miner（矿工）：参与挖矿的人或组织。
- Mining Machine（矿机）：专门为数字货币挖矿而设计的设备，包括基于软件、GPU、FPGA、专用芯片等多种实现。
- Mining Pool（矿池）：采用团队协作方式来集中算力进行挖矿，对产出的数字货币进行分配。
- PoS：Proof of Stake，股份持有证明，拥有代币或股权越多的用户，挖到矿的概率越大。
- PoW：Proof of Work，工作量证明，在一定难题前提下求解一个 SHA256 的 Hash 问题。
- Smart Contract（智能合约）：运行在区块链上的提前约定的合同；
- Sybil Attack（女巫攻击）：少数节点通过伪造或盗用身份伪装成大量节点，进而对分布式系统系统进行破坏。

超级账本相关术语

- Anchor（锚定）：一般指作为刚启动时候的初始联络元素或与其它结构的沟通元素。如刚加入一个通道的节点，需要通过某个锚点节点来快速获取通道内的情况（如其它节点的存在信息）。
- Auditability（审计性）：在一定权限和许可下，可以对链上的交易进行审计和检查。
- Block（区块）：代表一批得到确认的交易信息的整体，准备被共识加入到区块链中。
- Blockchain（区块链）：由多个区块链接而成的链表结构，除了初始区块，每个区块头部都包括前继区块内容的 Hash 值。
- Chaincode（链码）：区块链上的应用代码，扩展自"智能合约"概念，支持 Golang、Nodejs 等语言，多为图灵完备。
- Channel（通道）：Fabric 网络上的私有隔离机制。通道中的链码和交易只有加入该通道的节点可见。同一个节点可以加入多个通道，并为每个通道内容维护一个账本。
- Committer（提交节点）：一种 Peer 节点角色，负责对 Orderer 排序后的交易进行检查，选择合法的交易执行并写入存储。
- Commitment（提交）：提交节点完成对排序后交易的验证，将交易内容写到区块，并更新世界状态的过程。
- Confidentiality（保密）：只有交易相关方可以看到交易内容，其它人未经授权则无法看到。
- Endorser（推荐节点或背书节点）：一种 Peer 节点角色，负责检验某个交易是否合法，

是否愿意为之背书、签名。

- Endorsement：背书过程。按照链码部署时候的背书策略，相关 Peer 对交易提案进行模拟和检查，决策是否为之背书。如果交易提案获得了足够多的背书，则可以构造正式交易进行进一步的共识。
- Invoke（调用）：一种交易类型，对链码中的某个方法进行调用，一般需要包括调用方法和调用参数。
- Ledger（账本）：包括区块链结构（带有所有的交易信息）和当前的世界状态（world state）。
- Member（成员）：代表某个具体的实体身份，在网络中拥有自己的根证书。节点和应用都必须属于某个成员身份。同一个成员可以在同一个通道中拥有多个 Peer 节点，其中一个为 Leader 节点，代表成员与排序节点进行交互，并分发排序后的区块给属于同一成员的其它节点。
- MSP（Member Service Provider，成员服务提供者）：抽象的实现成员服务（身份验证，证书管理等）的组件，实现对不同类型的成员服务的可拔插支持。
- Orderer（排序节点）：共识服务角色，负责排序看到的交易，提供全局确认的顺序。
- Permissioned Ledger（带权限的账本）：网络中所有节点必须是经过许可的，非许可过的节点则无法加入网络。
- Privacy（隐私保护）：交易员可以隐藏交易的身份，其它成员在无特殊权限的情况下，只能对交易进行验证，而无法获知身份信息。
- System Chain（系统链）：由对网络中配置进行变更的配置区块组成，一般可以用来作为组成网络成员们形成的联盟约定。
- Transaction（交易）：执行账本上的某个函数调用或者部署、更新链码。调用的具体函数在链码中实现。
- Transactor（交易者）：发起交易调用的客户端。
- World State（世界状态）：即最新的全局账本状态。Fabric 用它来存储历史交易发生后产生的最新的状态，可以用键值或文档数据库实现。

常见问题

通用问题

问：区块链是谁发明的，有什么特点？

答：区块链相关的思想最早是比特币的发明者-中本聪（化名）在论文中提出（但没有明确定义）作为比特币网络的核心支持技术。自那以后，区块链技术逐渐脱离比特币网络，成为一种通用的可以支持分布式记账能力的底层技术，具有去中心化和加密安全等特点。

问：区块链和比特币是什么关系？

答：比特币是基于区块链技术的一种数字现金（cash）应用；区块链技术最早在比特币分布式系统中得到应用和验证，确保了比特币系统在 2009 年上线后，在完全自治情况下可以正常运转。

问：区块链和分布式数据库是什么关系？

答：两者定位完全不同。分布式数据库是解决高可用和可扩展场景下的数据存储问题；区块链则是在多方（无须中心化中介角色存在）之间提供一套可信的记账和合约履行机制。

问：区块链有哪些种类？

答：根据部署场景公开程度，可以分为公有链（Public Chain）、联盟链（Consortium Chain）和私有链（Private Chain）；从功能上看，可以分为以支持数字货币为主的数字货币区块链（如比特币网络）、支持智能合约的通用区块链（如以太坊网络）、面向复杂商业应用场景支持权限管理的分布式账本平台（如超级账本）。

问：（公有链情况下）区块链是如何保证没有人作恶的？

答：区块链并没有试图保障每一个人都不作恶，每个参与者都默认在最长的链上进行扩展。当某个作恶者尝试延续一个非法链的时候，实际上在跟所有的“非作恶”者进行竞争。因此，当作恶者超过一半（还要保持选择一致）时，在概率意义上才能破坏规则。而代价是一旦延续失败，所有付出的资源（例如算力）都将浪费掉。

问：区块链的智能合约应该怎么设计？

答：首先，智能合约类似其它应用程序，在架构上即可以采取 monolithic 的方式（一个合约针对一个具体商业应用，功能完善而复杂），也可以采取 microservice 的方式（即每个合约功能单一，多个合约一起构建应用）。 选择哪种模式根本上取决于其上商业应用的特点。从灵活性角度，推荐适当对应用代码进行切分，划分到若干个智能合约，尽量保持智能合约的可复用性。

问：如何查看 **PEM** 格式证书内容？

答：可以通过如下命令转换证书内容进行输出：`openssl x509 -noout -text -in <ca_file>`；例外，还可以通过如下命令来快速从证书文件中提取所证明的公钥内容：`openssl x509 -noout -pubkey -in <ca_file>`。

问：已知私钥，如何生成公钥？

答：对于椭圆曲线加密算法，可以通过如下命令生成公钥：`openssl ec -pubout -outform PEM -in <private_key>`。

问：如何校验某证书是否被根证书签名？

答：已知根证书文件 和待验证证书文件 情况下，可以使用如下命令进行验证：`openssl verify -CAfile <root_cafile> <ca_to_verify>`。

问：为何 **Hash** 函数将任意长的文本映射到定长的摘要，很少会发生冲突？

答：像 SHA-1 这样的 Hash 函数可以将任意长的文本映射到相对很短的定长摘要。从理论上讲，从一个很大的集合映射到一个小的集合上必然会出现冲突。Hash 函数之所以很少出现冲突的原因在于虽然输入的数据长度可以很大，但其实人类产生的数据并非全空间的，这些数据往往是相对有序（低熵值）的，实际上也是一个相对较小的集合。当然，这个集合自身可能比输出的结果要大，但这个冲突的概率远没有输入是全空间集合时那么夸张。

比特币、以太坊相关

问：比特币区块链为何要设计为每 **10** 分钟才出来一个块，快一些不可以吗？

答：这个主要是从公平的角度，当某一个新块被计算出来后，需要在全球的比特币网络内公布。临近的矿工将最先拿到消息并开始新一轮的计算，较远的矿工则较晚得到通知。最坏情况下，可能需要数十秒的延迟。为尽量确保矿工们都处在同一起跑线上，这个时间不能太短。但太长了又会导致每个交易的“最终”确认时间过长，目前看，10 分钟左右是一个相对合适的折中。另外，也是从存储代价的角度，让拥有不太大存储的普通节点可以参与到网络的维护。

问：比特币区块链每个区块大小为何是 **1 MB**，大一些不可以吗？

答：这个也是折中的结果。区块产生的平均时间间隔是固定的 10 分钟，大一些，意味着发生交易的吞吐量可以增加，但节点进行验证的成本会提高（Hash 处理约为 100 MB/s），同时存储整个区块链的成本会快速上升。区块大小为 1 MB，意味着每秒可以记录 1 MB/(10*60)=1.7 KB 的交易数据，而一般的交易数据大小在 0.2 ~ 1 KB。

实际上，之前比特币社区也曾多次讨论过改变区块大小的提案，但都未被最终接受。

问：以太坊网络跟比特币网络有何关系？

答：以太坊网络所采用的区块链结构，源于比特币网络。基于同样设计原理上，以太坊提出了许多改善设计，包括支持更灵活的智能合约、支持除了 PoW 之外的更多共识机制（尚未实现）等。

超级账本项目

问：超级账本项目与传统公有区块链有何不同？

答：超级账本是首个面向联盟链场景的开源项目，在这种场景下，参与账本的多方存在一定的信任前提，并十分看重对接入账本各方的权限管理、审计功能、传输数据的安全可靠等特性。超级账本在考虑了商业网络的这些复杂需求后，提出了创新的架构和设计，是首个在企业应用场景中得到大规模部署和验证的开源项目。

问：区块链最早是公有链形式，为何现在联盟链在很多场景下得到更多推崇？

答：区块链技术出现以前，中心化的信任机制可以实现很高的性能和便捷的监管，但一旦中心机制出现故障，则导致系统的信任前提发生破坏。区块链技术可以提供无中介化情况下的信任保障。公有链情况下，任何人都可以参与监督，可以实现信任的最大化，但随之而来带来包括性能低下、缺乏监管等问题。

联盟链在两者之间取得了平衡。非中心化的联盟共识，让系统可信任度以指数形式增加；同时，联盟形成的信任前提，可以在不影响信任的情况下实现更优化的性能，并支持权限管理。这对复杂应用场景特别企业场景可以提供更好的支持。

问：采用 **BFT** 类共识算法时，节点掉线后重新加入网络，出现无法同步情况？

答：这是某些算法设计导致的情况。掉线后的节点重新加入到网络中，其视图（View）会领先于其它节点。其它节点正常情况下不会发生视图的变更，发生的交易和区块内容不会同步到掉线节点。出现这种情况，可以有两种解决方案：一个是强迫其它节点出现视图变更，例如也发生掉线或者在一段时间内强制变更；另一种情况是等待再次产生足够多的区块后触发状态追赶。

问：超级账本 **Fabric** 里的安全性和隐私性是如何保证的？

答：首先，Fabric 1.0 及往后的版本提供了对多通道的支持，不同通道之间的链码和交易是不可见的，即交易只会发送到该通道内的 Peer 节点。此外，在进行背书阶段，客户端可以根据背书策略来选择性的发送交易到通道内的某些特定 Peer 节点。更进一步的，用户可以对交易的内容进行加密（基于证书的权限管理）或 Hash 处理，同时，只有得到授权的节点或用户才能访问到交易。另外，排序节点无须访问到交易内容，因此，可以选择不将完整交易（对交易输入数据进行隐藏，或者干脆进行加密或 Hash 处理）发送到排序节点。最后，所有数据在传输过程中可以通过 TLS 来进行安全保护。许多层级的保护需要配合使用来获得不同层级的安全性。

实践过程中，也需要对节点自身进行安全保护，通过防火墙、IDS 等防护对节点自身的攻击；另外可以通过审计和分析系统对可疑行为进行探测和响应。

相关企业和组织

排名不分先后，大部分信息来自互联网，不保证信息准确性，如有修改意见，欢迎联系。

国际

企业

- IBM: 贡献区块链平台代码到 HyperLedger 项目，推动区块链产业发展，跟多家银行和企业进行区块链项目合作。
- DTCC:贡献区块链代码到 HyperLedger 项目。
- Circle：基于区块链的支付应用公司，已获得 6000 万美元 D 轮投资，投资者包括 IDG、百度、中金甲子、广大投资等，目前年交易额超过 10 亿美金。
- Consensus：区块链创业团队，试图打造区块链平台技术和应用支撑，获得多家投资。

组织

- R3 CEV：创立于 2015 年 9 月，总部位于纽约的金融联盟组织，专注于研究和评估基于区块链的金融技术解决方案，由 40 多家国际金融机构组成，包括 Citi、BOA、高盛、摩根、瑞银、IBM、微软等。R3 开源技术已经 宣布 加入 HyperLedger 项目。
- [HyperLedger 社区]（https://hyperledger.org）：创立于 2015 年 12 月的技术社区，由 Linux 基金会管理，包括 IBM、Accenture、Intel、J.P.Morgan、R3、DAH、DTCC、FUJITSU、HITACHI、SWIFT、Cisco 等多家企业参与成立，试图打造面向企业应用场景的分布式账本平台。
- Ethereum 社区: 围绕以太坊区块链平台的开放社区。
- DAO： Distributed Autonomous Organization，基于以太坊平台的公募基金(众筹)组织，或去中心化的风投。众筹资金超过 1.6 亿美金。

国内

学术界

- 清华大学
- 中科院
- 上海交通大学

企业

- 中国电信：研究区块链相关技术，包括去中心化共享经济平台等。
- 世纪互联：投资区块链技术团队，牵头成立"中关村区块链产业联盟"。
- 银联：关注区块链相关技术，尝试引入基于区块链的银行业积分系统。
- 能链：专注于能源产品相关的区块链应用。
- 恒生电子：2016 年牵头成立"金链盟"，希望通过区块链技术为金融行业提供更简单的产品。
- 布比：主要关注数字资产管理的技术型创业企业，区块链相关平台和产品。
- 小蚁：主要关注对资产和权益进行数字化，2014年于上海组建成立。
- 火币：国内较大的比特币交易代理平台。
- BeLink：关注保险行业积分系统，主要产品为数贝荷包。
- BitSe：主要产品为唯链（Vechain），面向物品防伪追踪、数字版权管理相关。
- 万向集团：投资多家区块链创业团队，致力于推动产业发展。

组织

- 中关村区块链产业联盟：2016 年 2 月 3 日成立于北京，由世纪互联联合清华大学、北京邮电大学等高校、中国通信学会、中国联通研究院等运营商，及集佳、布比网络等公司发起；
- ChinaLedger：2016 年 4 月成立于上海，成员包括中证机构间报价系统股份有限公司、中钞信用卡产业发展有限公司北京智能卡技术研究院、万向区块链实验室、浙江股权交易中心、深圳招银前海金融资产交易中心、厦门国际金融资产交易中心、大连飞创信息技术有限公司、通联支付网络服务股份有限公司、上海矩真金融信息服务有限公司、深圳瀚德创客金融投资有限公司、乐视金融等；
- 金融区块链合作联盟（金链盟）：2016 年 5 月 31 日成立于深圳，包括平安银行、恒生电子、京东金融、腾讯微众银行、华为、南方基金、国信证券、安信证券、招商证券、博时基金等 25 家公司与机构。

ProtoBuf 与 gRPC

ProtoBuf 是一套接口描述语言（Interface Definition Language，IDL），类似 Apache 的 Thrift。

相关处理工具主要是 protoc，基于 C++ 语言实现。

用户写好 `.proto` 描述文件，之后便可以使用 protoc 自动编译生成众多计算机语言（C++、Java、Python、C#、Golang 等）的接口代码。这些代码可以支持 gRPC，也可以不支持。

gRPC 是 Google 开源的 RPC 框架和库，已支持主流计算机语言。底层通信采用 HTTP2 协议，比较适合互联网场景。gRPC 在设计上考虑了跟 ProtoBuf 的配合使用。

两者分别解决不同问题，可以配合使用，也可以分开单独使用。

典型的配合使用场景是，写好 `.proto` 描述文件定义 RPC 的接口，然后用 protoc（带 gRPC 插件）基于 `.proto` 模板自动生成客户端和服务端的接口代码。

ProtoBuf

需要工具主要包括：

- 编译工具：protoc，以及一些官方没有带的语言插件；
- 运行环境：各种语言的 protobuf 库，不同语言有不同的安装来源；

语法类似 C++ 语言，可以参考 ProtoBuf 语言规范：https://developers.google.com/protocol-buffers/docs/proto。

比较核心的，`message` 是代表数据结构（里面可以包括不同类型的成员变量，包括字符串、数字、数组、字典……），`service` 代表 RPC 接口。变量后面的数字是代表进行二进制编码时候的提示信息，1~15 表示热变量，会用较少的字节来编码。另外，支持导入。

默认所有变量都是可选的（optional），repeated 则表示数组。主要 service rpc 接口接受单个 message 参数，返回单个 message。如下所示。

```
syntax = "proto3";
package hello;

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
  repeated int32 number=4;
}

service HelloService {
  rpc SayHello(HelloRequest) returns (HelloResponse){}
}
```

编译最关键的参数是输出语言格式参数，例如，python 为 `--python_out=OUT_DIR` 。

一些还没有官方支持的语言，可以通过安装 protoc 对应的 plugin 来支持。例如，对于 Go 语言，可以安装

```
$ go get -u github.com/golang/protobuf/{protoc-gen-go,proto} // 前者是 plugin；后者是 go
 的依赖库
```

之后，正常使用 `protoc --go_out=./ hello.proto` 来生成 hello.pb.go，会自动调用 `protoc-gen-go` 插件。

ProtoBuf 提供了 `Marshal/Unmarshal` 方法来将数据结构进行序列化操作。所生成的二进制文件在存储效率上比 XML 高 3~10 倍，并且处理性能高 1~2 个数量级。

gRPC

相关工具主要包括：

- 运行时库：各种不同语言有不同的安装方法，主流语言的包管理器都已支持。
- protoc，以及 gRPC 插件和其它插件：采用 ProtoBuf 作为 IDL 时，对 .proto 文件进行编译处理。

类似其它 RPC 框架，gRPC 的库在服务端提供一个 gRPC Server，客户端的库是 gRPC Stub。典型的场景是客户端发送请求，同步或异步调用服务端的接口。客户端和服务端之间的通信协议是基于 HTTP2 的 gRPC 协议，支持双工的流式保序消息，性能比较好，同时也很轻。

采用 ProtoBuf 作为 IDL，则需要定义 service 类型。生成客户端和服务端代码。用户自行实现服务端代码中的调用接口，并且利用客户端代码来发起请求到服务端。一个完整的例子可以参考 https://github.com/grpc/grpc-go/blob/master/examples/helloworld 。

以上面 proto 文件为例，需要执行时添加 gRPC 的 plugin：

```
$ protoc --go_out=plugins=grpc:. hello.proto
```

gRPC 更多原理可以参考官方文档：http://www.grpc.io/docs。

生成服务端代码

服务端相关代码如下，主要定义了 HelloServiceServer 接口，用户可以自行编写实现代码。

```
type HelloServiceServer interface {
        SayHello(context.Context, *HelloRequest) (*HelloResponse, error)
}

func RegisterHelloServiceServer(s *grpc.Server, srv HelloServiceServer) {
        s.RegisterService(&_HelloService_serviceDesc, srv)
}
```

用户需要自行实现服务端接口，代码如下。

比较重要的，创建并启动一个 gRPC 服务的过程：

- 创建监听套接字： `lis, err := net.Listen("tcp", port)` ；
- 创建服务端： `grpc.NewServer()` ；
- 注册服务： `pb.RegisterHelloServiceServer()` ；
- 启动服务端： `s.Serve(lis)` 。

```
type server struct{}

// 这里实现服务端接口中的方法。
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloReply, e
rror) {
    return &pb.HelloReply{Message: "Hello " + in.Name}, nil
}

// 创建并启动一个 gRPC 服务的过程：创建监听套接字、创建服务端、注册服务、启动服务端。
func main() {
    lis, err := net.Listen("tcp", port)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    s := grpc.NewServer()
    pb.RegisterHelloServiceServer(s, &server{})
    s.Serve(lis)
}
```

编译并启动服务端。

生成客户端代码

生成的 go 文件中客户端相关代码如下，主要和实现了 HelloServiceClient 接口。用户可以通过 gRPC 来直接调用这个接口。

```
type HelloServiceClient interface {
	SayHello(ctx context.Context, in *HelloRequest, opts ...grpc.CallOption) (*HelloResponse, error)
}

type helloServiceClient struct {
	cc *grpc.ClientConn
}

func NewHelloServiceClient(cc *grpc.ClientConn) HelloServiceClient {
	return &helloServiceClient{cc}
}

func (c *helloServiceClient) SayHello(ctx context.Context, in *HelloRequest, opts ...grpc.CallOption) (*HelloResponse, error) {
	out := new(HelloResponse)
	err := grpc.Invoke(ctx, "/hello.HelloService/SayHello", in, out, c.cc, opts...)

	if err != nil {
		return nil, err
	}
	return out, nil
}
```

用户直接调用接口方法：创建连接、创建客户端、调用接口。

```
func main() {
    // Set up a connection to the server.
    conn, err := grpc.Dial(address, grpc.WithInsecure())
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer conn.Close()
    c := pb.NewHelloServiceClient(conn)

    // Contact the server and print out its response.
    name := defaultName
    if len(os.Args) > 1 {
        name = os.Args[1]
    }
    r, err := c.SayHello(context.Background(), &pb.HelloRequest{Name: name})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    log.Printf("Greeting: %s", r.Message)
}
```

编译并启动客户端，查看到服务端返回的消息。

参考资源链接

论文

- L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” Commun. ACM, vol. 21, no. 7, pp. 558–565, 1978.
- M Pease, R Shostak, L Lamport. Reaching Agreement in the Presence of Faults. Journal of the ACM, 1980, 27(2): 228-234.
- M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of Distributed Consensus with One Faulty Process,” J. ACM, vol. 32, no. 2, pp. 374–382, 1985.
- L. Lamport, “The Part-Time Parliament,” ACM Trans. Comput. Sys-tems, vol. 16, no. 2, pp. 133–169, 1998.
- M. Castro and B. Liskov, “Practical Byzantine Fault Tolerance,” Proc. Symp. Oper. Syst. Des. Implement., no. February, pp. 1–14, 1999.
- 中本聪，"Bitcoin: A Peer-to-Peer Electronic Cash System（比特币：一种点对点的电子现金系统）"，https://bitcoin.org/bitcoin.pdf，2008. A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, “Enabling Blockchain Innovations with Pegged Sidechains,” pp. 1–25, 2014.
- T. D. Joseph Poon, “The Bitcoin Lightning Network: Scalable Off-Chain Payments, http://lightning.network/lightning-network-paper.pdf,” pp. 1–59, 2016.
- Gentry C., Halevi S.，"Implementing Gentry’s Fully-Homomorphic Encryption Scheme". In: Paterson K.G. (eds) Advances in Cryptology – EUROCRYPT 2011. EUROCRYPT 2011. Lecture Notes in Computer Science, vol 6632. Springer, Berlin, Heidelberg.
- van Dijk M., Gentry C., Halevi S., Vaikuntanathan V., "Fully Homomorphic Encryption over the Integers". In: Gilbert H. (eds) Advances in Cryptology – EUROCRYPT 2010. EUROCRYPT 2010. Lecture Notes in Computer Science, vol 6110. Springer, Berlin, Heidelberg.
- López-Alt, Adriana, Eran Tromer, and Vinod Vaikuntanathan. "On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption.". Proceeding STOC '12 Proceedings of the forty-fourth annual ACM symposium on Theory of computing, Pages 1219-1234.
- I. Miers, C. Garman, M. Green, and A. D. Rubin, “Zerocoin: Anonymous distributed e-cash from bitcoin,” Proc. - IEEE Symp. Secur. Priv., pp. 397–411, 2013.
- F. Reid and M. Harrigan, “An analysis of anonymity in the bitcoin system,” Secur. Priv. Soc. Networks, pp. 197–223, 2013.
- K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, “Formal Verification of Smart Contracts,” 2016.

项目网站

- 比特币项目官方网站：https://bitcoin.org/。
- blockchain.info：比特币信息统计网站。
- bitcoin.it：比特币 wiki，相关知识介绍。
- 以太坊项目官方网站：https://www.ethereum.org。
- 以太坊网络的状态统计：https://etherchain.org/
- 超级账本项目官方网站：https://hyperledger.org;
- 超级账本 Docker 镜像：https://hub.docker.com/r/hyperledger/。

培训课程

- Bitcoin and Cryptocurrency Technologies, Princeton University。

区块链即服务

- IBM Bluemix BaaS： https://console.ng.bluemix.net/catalog/services/blockchain/。
- 微软 Azure BaaS：https://azure.microsoft.com/en-us/solutions/blockchain。